Youpi Handbook

M. Monnerville and G. Semah

November 10, 2008

Youpi Handbook
by M. Monnerville and G. Semah

Contents

List of Figures

List of Tables

Abstract

Welcome to Youpi! This handbook covers the installation and day to day use of Youpi. Please note that this manual is a work in progress. As such, some sections may be incomplete.

Preface

Youpi (stands for YOUpi is your processing PIpeline) is a web application providing high level functionnalities to perform data reduction on scientific **FITS** images.

Organization of This Book

Getting This Book

Request for Comments

Part I

Getting Started

Chapter 1

Installing Youpi

At least one chapter, reference, part, or article is required in a book.

1.1 Getting Youpi

1.2 Software dependencies

1.2.1 Mandatory packages

1.2.2 Optional packages

1.3 Installation

1.4 Configuration

1.5 Testing Your Installation

1.6 Upgrading to New Releases

Chapter 2

Using Youpi

TOTO

2.1 Pre-ingestion of FITS Tables

2.2 FITS Data Ingestion

2.3 Processing Your Data

2.4 Processing Results

2.5 Live Monitoring of Processings

2.6 The Shopping Cart

Part II

Youpi Administration

Chapter 3

Common Tasks

Part III

Dive into Youpi

Chapter 4

Merise Analysis

This is Merise Analysis.

4.1 Data Dictionnaries

4.1.1 Image Entity

4.1.2 FITSin Entity

4.1.3 Scamp Entity

4.1.4 Astrophoto Calibration Entity

4.1.5 Calibration Kit Entity

4.1.6 Instrument Entity

4.1.7 Channel Entity

4.2 The Conceptual Model of Data

Properties surrounded with brackets are entities identifiers. The `User` entity (and its relations) is displayed with a dashed style because it is part of the standard Django's database model.

Table 4.1 Data dictionnary of Image entity

Full name	**DB field name**	**Description**	**Unit**	**Type**	**Display Format**
Sky footprint	`skyfootpr-int`	Footprint of image on sky	deg	Multi-polygon	`%8f`
Image name	`name`	Image name (without the .fits extansion)	-	string	`%s`
Image path	`path`	Path of image file (cluster or local)	-	string	`%s`
Right ascension	`alpha`	Right ascension of field centre	deg	double	`%02d:%02d-:%05.2f`
Declination	`delta`	Declination of field centre	deg	double	`%+02d:%02-d:%04.1f`
Equinox	`equinox`	Equinox at time of observation	yr	double	`%7.2f`
Object name	`object`	Object identifiant	-	string	`%s`
Observation date	`dateobs`	Date and time at start of observation	-	datetime	`date %c format`
Exposure time	`exptime`	Effective exposure time	s	double	`%9.2f`
Magnitude zero-point	`photc`	Magnitude Zero-point for 1s exposure	mag	float	`%+8.4f`
Extinction coefficient	`photk`	Extinction coefficient at airmass 1	mag	float	`%7.4f`
Airmass	`airmass`	Airmass at start of observation	-	float	`%8.4f`
Absorption	`absorption`	Absorption at start of observation	mag	float	`%+7.4f`
Checksum	`checksum`	Image file checksum	-	unsigned long long	`%0x`
Gain	`gain`	Detector conversion factor	e-/ADU	vector of floats	`%8.2f`
Ingestion date	`ingestion-_date`	Date and time at start of ingestion	-	datetime	`date %c format`
Flatfield	`flat`	Flatfield filename	-	string	`%s`
Mask	`mask`	Mask filename	-	string	`%s`
Ds9 region file	`reg`	Ds9 region filename	-	string	`%s`
Validation flag	`QSOstatus`	Image validation status	-	unsigned char	`%c`

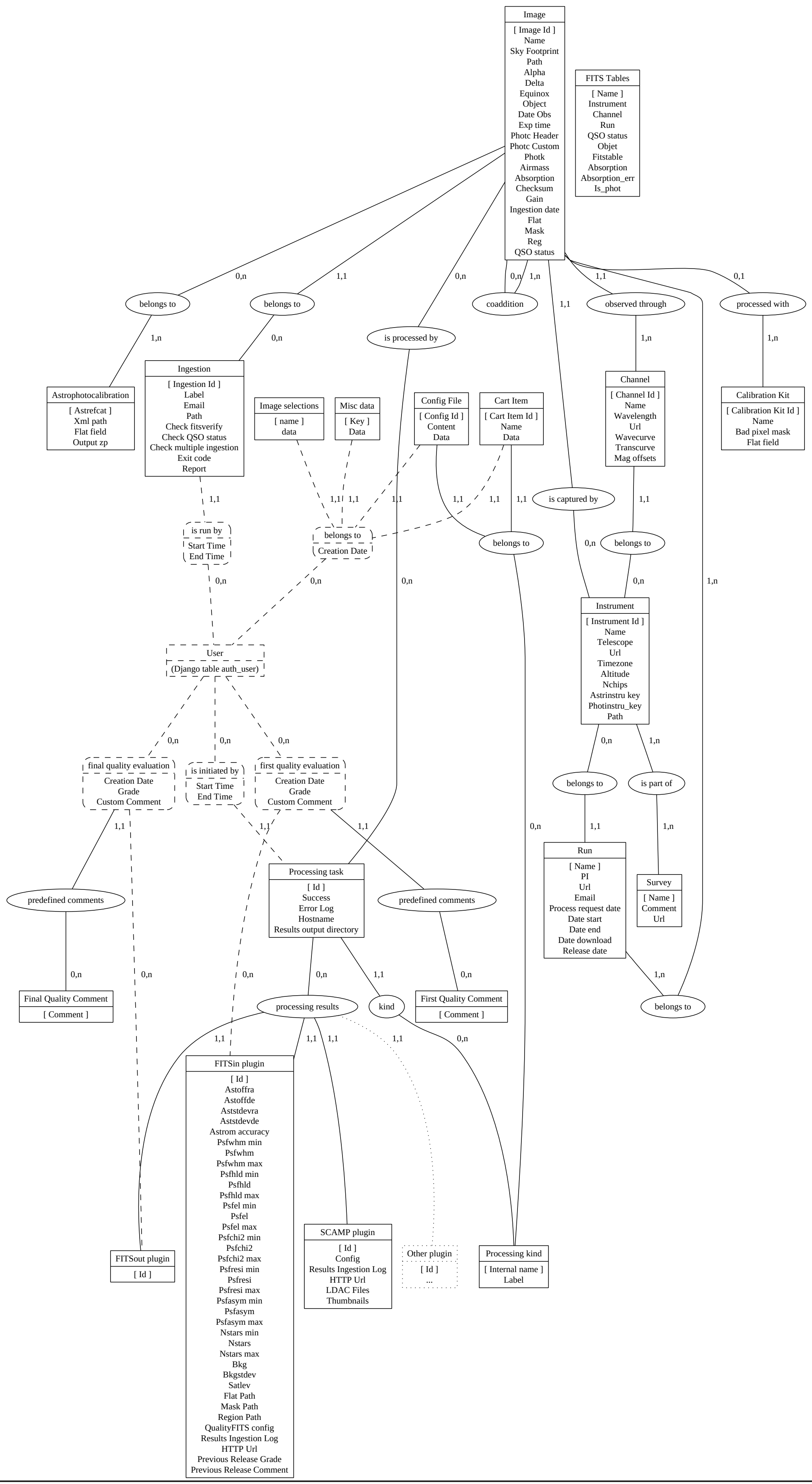
Table 4.2 Data dictionnary of FITSin entity

Full name	DB field name	Description	Unit	Type	Display Format
RA offset	`astoffra`	Offset wrt astrometric reference catalogue in RA	arcsec (")	float	`%8.3g`
Dec offset	`astoffde`	Offset wrt astrometric reference catalogue in Dec	arcsec (")	float	`%8.3g`
	`astromacc-uracy`				
RA std dev	`aststdevra`	Dispersion wrt astrometric reference catalogue in RA	arcsec(")	float	`%8.3g`
Dec std dev	`aststdevde`	Dispersion wrt astrometric reference catalogue in Dec	arcsec(")	float	`%8.3g`
Minimum PSF FWHM	`psffwhmmin`	Minimum Full-Width at Half-Maximum of the PSF	arcsec(")	float	`%8.3g`
Average PSF FWHM	`psffwhm`	Central/average Full-Width at Half-Maximum of the PSF	arcsec(")	float	`%8.3g`
Maximum PSF FWHM	`psffwhmmax`	Maximum Full-Width at Half-Maximum of the PSF	arcsec(")	float	`%8.3g`
Minimum PSF half-light diameter	`psfhldmin`	Minimum half-light diameter of the PSF	arcsec(")	float	`%8.3g`
Average PSF half-light diameter	`psfhldm`	Average half-light diameter of the PSF	arcsec(")	float	`%8.3g`
Maximum PSF half-light diameter	`psfhldmax`	Maximum half-light diameter of the PSF	arcsec(")	float	`%8.3g`
Minimum PSF elongation	`psfelmin`	Minimum elongation of the PSF	-	float	`%5.2f`
Average PSF elongation	`psfel`	Central/average elongation of the PSF	-	float	`%5.2f`
Maximum PSF elongation	`psfelmax`	Maximum elongation of the PSF	-	float	`%5.2f`
Minimum		Minimum			

Table 4.3 Data dictionnary of Scamp entity

Full name	**DB field name**	**Description**	**Unit**	**Type**	**Display Format**
Results ingestion log	`log`	Results ingestion log	-	string	`%s`
Scamp configuration	`config`	Scamp configuration file serialized content (base64 encoding over zlib compression)	-	string	`%s`
HTTP url	`www`	URL to Scamp output data	-	string	`%s`
LDAC Files	`ldac_files`	List of LDAC files used during processing (serialized data)	-	list	`%s`
Thumbnails	`thumbnails`	True if thumbnails of output images (only group #1) have been created during processing (with convert utility)	-	boolean	`%s`

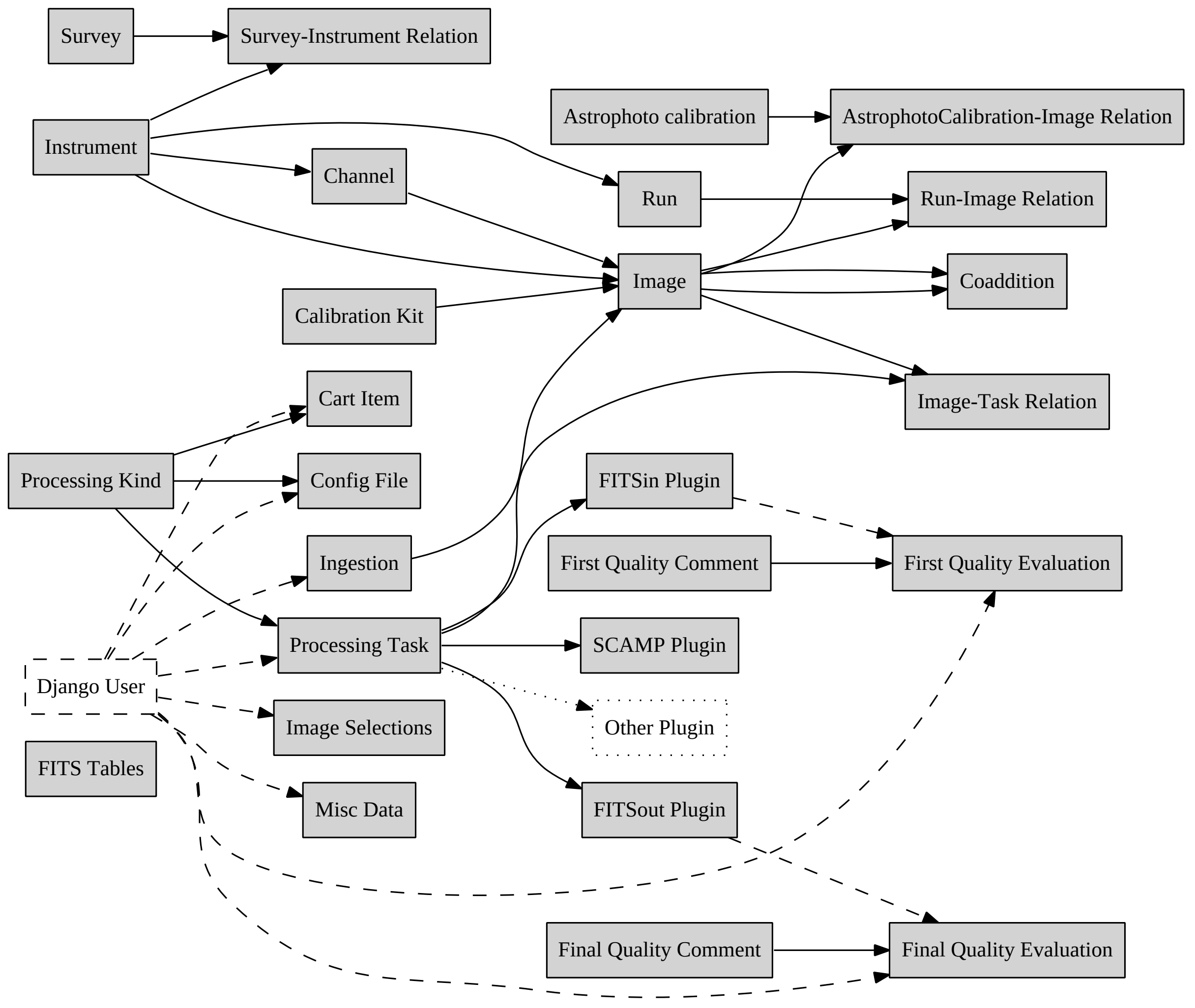
Figure 4.1 The conceptual model of data (MCD)



4.3 The Codasyl Logical Model

The codasyl logical model resulting of transformation rules applied to the previous MCD model. The Django User record is displayed with a dashed style because it is part of the standard Django's database model.

Figure 4.2 The codasyl Logical Model



4.4 The Relationnal Logical Model of Data

Chapter 5

Writing Plugins for Youpi

This section is focusing on the processing capabilities of Youpi and aims to be a short guideline - by providing useful background information - to the software developper that needs to understand, improve and write processing plugins for Youpi.

5.1 Plugin Concept Overview

Plugins are computer programs that interact with a host application to provide specific on-demand new features. In order to be a rather generic pipeline for processing astronomical data, Youpi - which is powered by Django, an open source web application framework that uses the Python programming language - has been designed to allow custom programs to operate a wide range of processings on data.

For this purpose, Youpi comes with an easy and convenient API allowing third-party developpers to extend its standard functionalities.

Basically, a Youpi processing plugin is made of four distinct files. One of them is a Python script used for all server-side actions, such as (but not limited to) database interactions, variables or constants you would like Django to substitute in your HTML templates, and class member methods you would like to be available to the public interface of your plugin (so you can call them using AJAX calls from your Javascript files). The remaining three files are client-side related: a Javascript file holding all your client-side routines and two HTML templates files for rendering the plugin on the processing page and rendering the plugin related items in the shopping cart respectively.

Finally, when you are done writting your new plugin, the plugin manager will find, register and load your code into Youpi's processing part. Let's begin with a simple example.

5.2 Hello, world!

Todo...

5.3 Naming Conventions

Before we continue with a concrete example in the next section, let us have a brief discussion about naming policy related to Youpi plugin development. All your plugins related files should live in two places (only those paths will be searched for file inclusion):

- The `templates` directory, which should hold all your HTML and JavaScript client-side template files. Let *myplugin* be your new plugin's name. In order to respect Youpi naming policy, you should prepend file names with the '`plugin_`' keyword (note the underscore). So, you should create and use the following files:

  ```
  templates/plugin_myplugin_item_cart.html
  templates/plugin_myplugin.html
  templates/plugin_myplugin.js
  ```

  Files in this directory get substituted by Django before client-side rendering. One might wonder why every plugin's JavaScript file is in there and why it needs to be substituted. The reason for this rather unexpected location (it should preferably belong to the `media/js` directory, defined to be the location holding all JavaScript-related files) is that server-side Django substitution may help with the lack of robust namespacing support in JavaScript. Let us consider the rendering of the HTML processing page, as defined in the `templates/processing.html` file. Two parts are of interest:

Example 5.1 HTML rendering of the processing page

```
<script type="text/javascript">
  {# Load custom plugin's javascript code if available #}
  {% for plugin in plugins %}
  {% include plugin.jsSource %} ❶
  {% endfor %}
</script>
...
{% for plugin in plugins %}
{% if forloop.first %}
<div id="menuitem_{{ plugin.id }}">
  {% else %}
<div id="menuitem_{{ plugin.id }}" style="display: none">
  {% endif %}
  <table width="100%">
    <tr>
      <td>
        <p>Plugin description: {{ plugin.description }}, version {{ plugin.version }}</p>
        <div align="center">{% include plugin.template %}</div> ❷
      </td>
    </tr>
  </table>
</div>
{% endfor %}
```

❶ Includes each registered plugin's javascript code. The `include` statement is a Django's built-in template tag which loads a template file and renders it *with the current context*, thus making the `plugin` variable available into your JavaScript `jsSource` file too.

❷ Shows how plugin HTML template data is included into the final HTML document.

Since the `plugin` variable is a Python object (which inherits the `Spica2Plugin` class), Django's template system is able to access all its data members. Context variable lookup can get around this namespacing issue by, for example, prefixing function and variable names with the plugin's unique identifier (See id variable definition). Therefore, instead of writing this:

```
function my_plugin_function() {}
var my_global_var;
<div id="my_unique_id"></div>
```

do write instead:

```
function {{ plugin.id }}_my_plugin_function() {}
var {{ plugin.id }}_my_global_var;
<div id="{{ plugin.id }}_my_unique_id"></div>
```

Let `myid` be your plugin's unique identifier. After rendering, final HTML data will look like this:

```
function myid_my_plugin_function() {}
var myid_my_global_var;
<div id="myid_my_unique_id"></div>
```

- The `spica2/plugins/` directory, which should hold your server-side Python ascript. There is no particular naming policy since all Python files that belongs to this directory are parsed by the plugin manager at registration time (see [?para] [19]). For example, default plugin Python filenames are among the following:

```
spica2/plugins/qualityfitsin.py
spica2/plugins/qualityfitsout.py
spica2/plugins/sextractor.py
spica2/plugins/scamp.py
spica2/plugins/swarp.py
```

5.4 About AJAX calls

For enhanced client-server interactions, you will certainly want to use the usefull `XMLHttpRequest` facility provided by JavaScript. Sending asynchronous queries to a server allows to do powerful things inside your client scripts, such as manipulating the DOM [1] according to the response received by the `XMLHttpRequest` query.

If you are dealing with AJAX calls in Youpi, your are advised to use the simple `HttpRequest` object defined in file `media/js/xhr.js` that works as a wrapper with convenient methods. It provides a very simple API with some useful callback mechanisms for error and result handling. As you can see in Example 5.2, its use is rather straightforward:

Example 5.2 Basic `HttpRequest` object use

```
<script type="text/javascript" src="/media/js/xhr.js"></script>
<div id="container"></div> ❶

<script type="text/javascript">
  var r = new HttpRequest(
    'container', ❷
    null, ❸
    function(resp) { ❹
      var data = resp['result']; ❺
    }
  );

  var post = 'Key1=' + value1 + '&Key2=' + value2; ❻
  r.send('/url/to/server/script/', post); ❼
</script>
```

❶ Sets an HTML `div` element which acts like a block container. It is only used to display some kind of 'Data loading, please wait...' message while waiting for incoming data. [2]

❷ Id of the DOM container. This parameter can be a DOM unique identifier or a DOM object.

❸ The `null` parameter indicates that no custom error handler is defined, so the default one (displaying an error message embedded into the `div` container) will be used instead.

❹ The last argument is about handling results. You should define a callback function with only one argument. This function prototype has to be used every time you want to access your results. Once the `HttpRequest` object gets a successful response from the server, it passes an evaluation of the returned data to your custom handler code. Your data, if any, will always be accessible through the content of `resp['result']` and can be of any supported JavaScript type.

❺ Handle your response here.

❻ Defines some parameters to be sent as POST data.

❼ Finally, call the `send()` method to send the POST HTTP query.

[1] The Document Object Model is fully implemented in recent versions of JavaScript and is supported in almost all today's web browsers

Youpi's `HttpRequest` object only supports POST HTTP requests; the `Content-type`'s request header is always set to `application/x-www-form-urlencoded`. Thus, you can submit optional POST data as a second parameter to its `send()` function, the only one that effectively issues the query.

When you are writing a processing plugin for Youpi, you have to write both client-side - your HTML template file(s) and your JavaScript `jsSource` file - and server-side code (your Python class that inherits `Spica2Plugin`). Because all your plugin's server-side code have to be a part of your Python class, Youpi provides a way to call a specific member method from your plugin's JavaScript code. A dedicated Django's URL has been defined and serves as a unique entry point, so that you can use it to access any member method of any available plugin. Have a look at its definition in `urls.py`:

```
(r'^spica2/process/plugin/$', 'processing_plugin')
```

This line explicitely maps the `/spica2/process/plugin/` URL to the server-side Django's callback function `processing_plugin()` defined in `views.py`:

```
def processing_plugin(request):
  try:
    pluginName = request.POST['Plugin'] ❶
    method = request.POST['Method'] ❷
  except Exception, e:
    return HttpResponseBadRequest('Incorrect POST data')

  plugin = manager.getPluginByName(pluginName) ❸
  try:
    res = eval('plugin.' + method + '(request)') ❹
  except AttributeError:
    raise PluginError, "Plugin '%s' has no method '%s'" % (plugin.id, method)

  # Response must be a JSON-like object
  return HttpResponse(str({'result' : res}), mimetype = 'text/plain') ❺
```

❶, ❷ `Plugin` and `Method` are required POST parameters. The former is the plugin's identifier (see id description), the latter your plugin's method member that is to be executed.

❸ Asks the plugin manager which plugin matches this `pluginName` string identifier; it raises a `PluginManagerError` Python exception if no plugin with this name has been found.

❹ Executes the requested method. As you can see, the `request` variable is passed to every plugin method so that you can access easily request parameters from your plugin code.

❺ Finally, the function returns a Python dictionary that matches the JSON[3] format.

The `processing_plugin()` function is defined as a Django *view function* which is a Python function that takes a web request - an `HttpRequest` object - as argument and returns a web response. The `request` variable is an `HttpRequest` instance that holds really useful information such as the HTTP method used in the request, GET or POST data parameters, all HTTP headers, session and currently logged-in user data.

The JSON response is well suited for transmitting structured data over a network connection and can be natively processed by JavaScript within your callback response handler:

```
function myHandler(resp) {
  var data = resp['result'];
  // Now data is (should be) a JSON object ready for processing
}
```

Thus, in order for your response to be parsed and processed successfully at client-side level, every plugin's method have to return a JSON-aware Python dictionary.

[3] JavaScript Object Notation data-interchange format

5.5 Active Monitoring Interface Integration

Youpi comes with a built-in job monitoring web interface, the *Active Monitoring Interface* (AMI for short), that allows realtime monitoring of all Youpi-related jobs running on the Condor cluster. Each entry (one entry per job) gives information about

- the *Condor job's ID* which identifies a job uniquely on the cluster
- the *Youpi's job owner* which is the username of the Youpi's account that initiated the job. Please note that this is *not* the same as the UNIX user that executes Condor jobs on the cluster. Youpi's job owner is the login name - stored in the database - of the registered account that submitted the job through the web interface.
- the kind of processing being made with a *short one-line description*
- the *remote cluster host* where the job is running
- *elapsed time* since job submission
- the *Condor job's status* on the cluster. A job can be marked as '`Idle`' if it is part of the queue, waiting for available resources (that matches its execution requirements) in order to run, or marked as '`Running`' if it is effectively executing on a node, or marked as '`Hold`' if Condor caught some exception while trying to terminate and release the job. A job that terminates with errors will not be part of Condor's queue anymore and will disappear from Youpi's AMI too. There will be *no error reporting* through the AMI. In such cases, the user will have to check Condor's error log files manually, which is not very user-friendly. Therefore, suitable error handling should be added to your code to catch exceptions and avoid this situation.

The AMI periodically monitors the XML output of the **condor_q** command. Almost all previously mentionned information is retreived while parsing the XML data except for the *Youpi's job owner* and the *short one-line description* fields, which are not Condor-related information but instead Youpi-related user data. Thus, in order to make your plugin support the AMI, special care must be taken when dealing with Condor submission file generation within your scripts.

Passing extra user data to Condor is achieve by defining the `SPICA_USER_DATA` environment variable in the Condor submission file. Its content has to be a base64-encoded serialized Python dictionary with at least the two mandatory keys `Descr` that stands for 'description' and `UserID`, which is the Django's unique user identifier (in fact, there is a third mandatory key, `Kind`, that we will discuss shortly):

```
userData = {
  'Descr' : "%s of" % self.optionLabel,
  'UserId': request.user.id,
  # This dictionary can contain any kind of Python data
  'OtherData' : []
}

# CSF generation
csf = """
executable  = mybin
universe    = vanilla
environment = SPICA_USER_DATA=%s
...
""" % base64.encodestring(marshal.dumps(userData)).replace('\n', '')

# Submit the job on Condor
pipe = os.popen("%s/condor_submit %s 2>&1" % (CONDOR_PATH, csf))
data = pipe.readlines()
pipe.close()
```

Also, note that the presence of the `SPICA_USER_DATA` environment variable ensures that only Youpi related jobs are filtered and monitored. Other Condor jobs will not be monitored by the AMI.

5.6 Wrap Your Cluster Code!

Every built-in Youpi plugin that initiates jobs on the Condor cluster keeps track of various processing information by storing it into the database. The spica2_processing_task MySQL table holds information about a job exit status, the user that initiated the job, the kind of processing performed, start and end times, the cluster node used for processing data and the complete error log content if success is null:

```
mysql> desc spica2_processing_task;
+------------+--------------+------+-----+---------+----------------+
| Field      | Type         | Null | Key | Default | Extra          |
+------------+--------------+------+-----+---------+----------------+
| id         | int(11)      | NO   | PRI | NULL    | auto_increment |
| success    | tinyint(1)   | NO   |     |         |                |
| user_id    | int(11)      | NO   | MUL |         |                |
| kind_id    | int(11)      | NO   | MUL |         |                |
| start_date | datetime     | NO   |     |         |                |
| end_date   | datetime     | NO   |     |         |                |
| error_log  | longtext     | YES  |     | NULL    |                |
| hostname   | varchar(255) | YES  |     | NULL    |                |
+------------+--------------+------+-----+---------+----------------+
```

If your plugin is designed to send jobs to the Condor cluster, you may find useful to use the wrapper_processing.py wrapper script provided with the distribution. Without it, it would be rather difficult to get accurate information about a processing task executing on a cluster's node. Moreover, getting the task's exit code would not be easy either as you would have to parse Condor log files.

The wrapper_processing.py script will help you by encapsulating your processing task. It is the one that will be executed on the target node, taking control over your processing task execution, performing the following actions:

- The wrapper script first performs some sanity checks. It looks for the Kind dictionary key into its first argument, which must be a *base64-encoded serialized Python dictionary* (remember the SPICA_USER_DATA variable in the previous section?). If not found, a WrapperError Python exception is raised, thus terminating the processing. This keyword is mandatory because it allows the wrapper script to take special action depending on Kind's associated value. Youpi's built-in plugins use the Plugin.id data member for the Kind keyword.

  In order to know if a *processing kind* is available, Youpi maintains a spica2_processing_kind table with some information about registered plugins [4]. Again, if Kind's associated value does not match any of that table's entries, a WrapperError Python Exception is raised; the wrapper script is aborted.

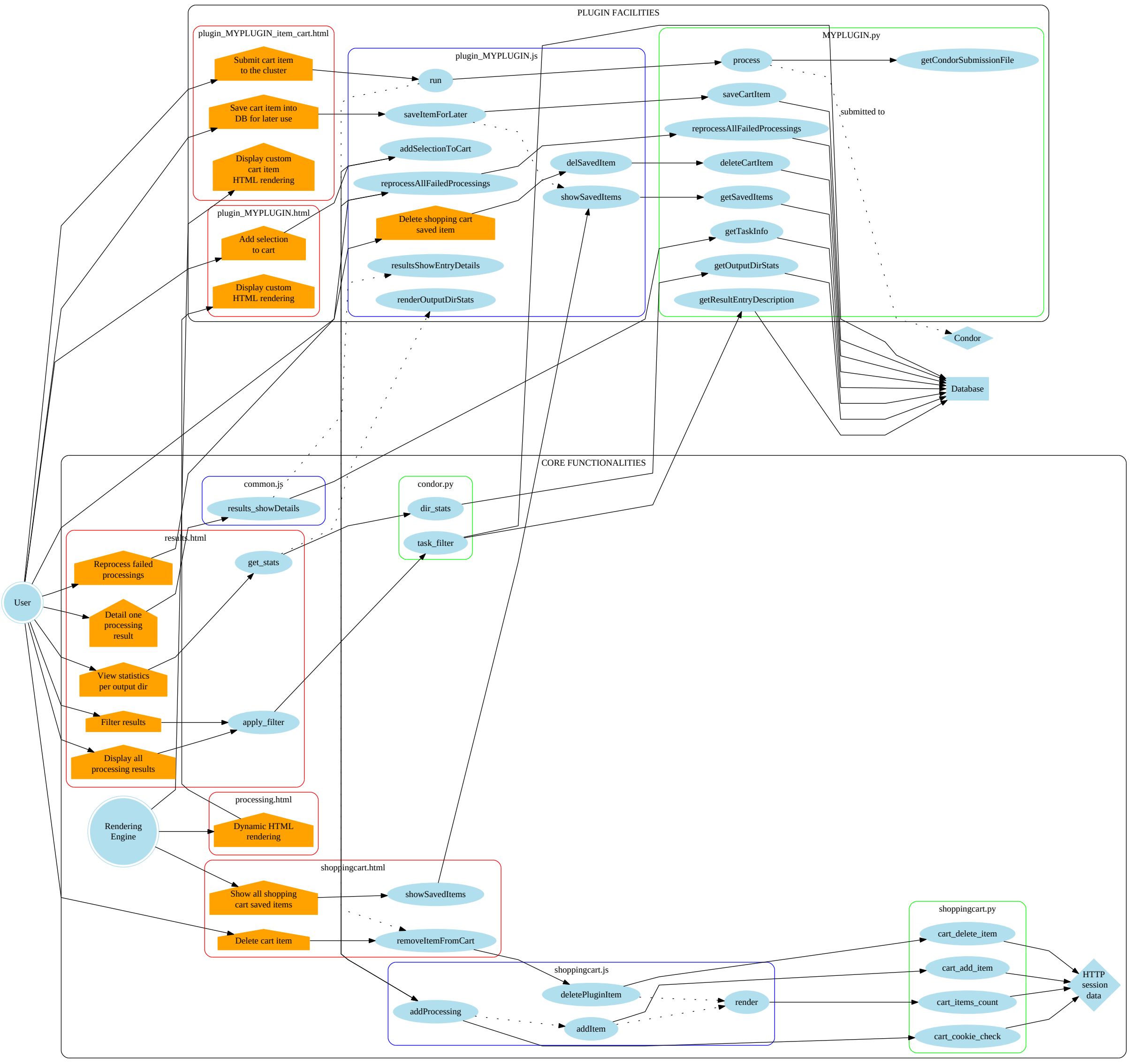
- The wrapper script then call its process() function, passing it the userData Python dictionary and the remaining command line arguments. This function

  1. inserts an entry in the spica2_processing_task table, filling the information related to the node's hostname, the Django's user ID, the processing kind and starting date.
  2. executes the command line arguments - your real processing stuff - in a subshell and waits for the shell to finish executing the command. It then saves the exit status of the shell.
  3. performs some custom actions depending on your Kind's keyword value. This is where you can enhance the wrapper script to *add support for your plugin*.
  4. updates the spica2_processing_task table with the error log content if the processing was not successful (content is compressed and base64-encoded), fills the end_date and success data fields.

- Once your processing is over, the wrapper script exits, the job terminates and is removed from Condor's queue.

[4] TODO: improve this part

Figure 5.1 Plugin function calls



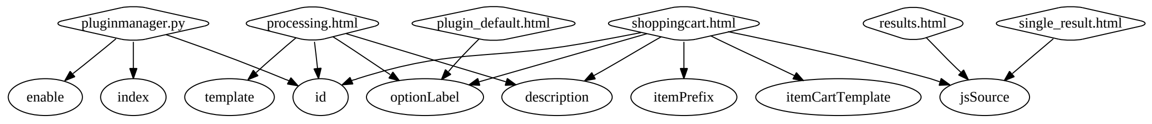
5.7 Mandatory Data Members

Before you can start writing your brand new plugin, you must be aware that some variables - your class' data members - are needed in order to make Youpi behave the way you expected.

For example, to prevent registration of non plugin-related code, some sanity checks are performed to ensure that imported modules are conforming to the expected plugin infrastructure. Those sanity checks need to access specific data members in every plugin's code, thus making some data members mandatory at plugin's registration time.

Other mandatory data members are accessed by core - mainly HTML templates - files. Figure 5.2 provides some information about those dependencies. Core Youpi files (with rounded diamond shape) need some Python data members, thus making their definition at plugin level mandatory.

Figure 5.2 Plugin data members dependencies



MANDATORY PYTHON DATA MEMBERS ARE

`enable` Boolean that states whether a plugin should be registered by the plugin manager, making it available for the entire application. Set `enable` to `False` if you want to disable a plugin.

Every plugin inherits the `Spica2Plugin` class (defined in `pluginmanager.py`) which defines an `enable` data member with a value that defaults to `True`.

`id` String that identifies a plugin uniquely. Two plugins can't have the same id. If some internal names collide, the plugin manager aborts plugins loading and raises a `PluginManagerError` Python exception. If you dive into some parts of the code of existing plugins, you may see that this `id` string is used quite intensively in templates[5] for prefixing variable or function names. This is useful when dealing with Javascript code since this technique allows to easily deal with the lack of robust native namespace support. The only file referencing the `id` variable is `pluginmanager.py`.

`index` Integer used by the plugin manager at loadtime to visually sort submenus (one submenu per plugin) on the processing web page. The plugin that comes with the smaller index gets displayed first (from left to right) on the screen. Note that two plugins may have the same index but this will have no effect on the resulting order (the sort algorithm is fairly simple and does not do anything special in that case). Again, the only file referencing the `index` variable is `pluginmanager.py`.

`optionLabel` String used by the plugin manager to set a plugin's menu item title on the processing page. Note that no particular checks are performed so that plugins with the same `optionLabel` value will have the same menu title, which is not what you might want. This variable is referenced by three template files: `processing.html`, `plugin_default.html` and `shoppingcart.html`.

`description` Short inline string used to display some plugin's general action. Youpi uses this variable to name the low-level program that really does the job. While the `optionLabel` variable should hold a rather general option name, the `description` data member should be used to define more accurate, lower level kind of information. This variable is referenced by the templates files `processing.html` and `shoppingcart.html`.

`itemPrefix` String used at the shopping cart level to add a custom prefix to items' names in the cart. The only file referencing this variable is the `shoppingcart.html` template.

`template` String used to specify the Django's template file to use for custom HTML rendering on the processing page. This file is the critical part of any plugin's client-side code. Since its content is processed by the Django's template rendering engine, any template tags and filters can be used within the file and, of course, server-side context variables are substituted before rendering. The only file referencing this variable is the `processing.html` template.

Please note that, technically speaking, the `template` data member is not mandatory. When Django renders the `processing.html` template, it tries to include - for every registered plugin - the HTML template referenced by the `template` plugin's data member. In `processing.html`, the (simplified) substitution code looks like the following:

```
{% for plugin in plugins %}
  <div>{% include plugin.template %}</div>
{% endfor %}
```

The `Spica2Plugin` class defines a generic `template` data member with a value that defaults to the file `plugin_default.html`. So if you forget to provide a `template` data member to your custom plugin, this default HTML template will be used instead, reminding you that you certainly missed something.

`itemCartTemplate` String defining a Django HTML template that will be included into the shopping cart template for custom rendering. This is where you decide what your plugin's related items will look like and behave on the shopping cart page. The only file referencing this variable is the `shoppingcart.html` template.

[5] Files whose content is dynamically substituted at runtime by Django.

`jsSource` String used to specify a filename containing some custom JavaScript source code for your plugin. Even if you don't have any JavaScript code to put into this file right now, it is a good practice to create an empty file and set up your `jsSource` data member accordingly. Indeed, core templates files such as `shoppingcart.html` *will* try to include every plugin's external JavaScript code:

```
{# Load custom plugin’s javascript code when needed #}
{% for plugin in plugins %}
  {% include plugin.jsSource %}
{% endfor %}
```

The `jsSource` variable is referenced by three template files: `shoppingcart.html`, `results.html` and `single_results.html`.

Part IV

Appendixes

Appendix A

Frequently Asked Questions (FAQ)

TODO

Appendix B

Resources

TODO

Appendix C

GNU Free Documentation License

Version 1.1, March 2000

> Copyright (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

C.1 PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

C.2 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text

C.3 VERBATIM COPYING

C.4 COPYING IN QUANTITY

C.5 MODIFICATIONS

C.6 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

C.7 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

C.8 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

C.9 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

C.10 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

C.11 FUTURE REVISIONS OF THIS LICENSE

C.12 How to use this License for your documents