

Web2py Plug-In Specification

Revision 4 - 2010-04-14

By: Thadeus Burgess and the web2py community

Introduction

Currently plug-ins for web2py is in active discussion and development. This paper serves as the current reference and standard of the web2py plug-in system, as well as containing current limitations and potential solutions. The goal for plug-ins is to make web2py development easier and easier.

Definition of a web2py plug-in

It is a system to provide functionalities that are common to most websites, while still allowing maximum control over system integration and interoperability.

A plug-in is a set of classes and procedures that define a framework that solves a common but narrow problem base. This includes plug-ins that provide common extensions (just database models), and plug-ins that can override base web2py functionality. A plug-in is a first class controllers, models, views, modules, static files that acts as a sub-set of current web2py application implementations.

Problem space for a plug-in.

Plug-ins will provide extensible data structures surrounding a problem. Examples of current problem spaces plug-ins could solve.

1. Comments
2. Ratings
3. Blog
4. Cached Javascript Minification
5. Search
6. Tagging
7. Meta / SEO
8. Event System / Notifications
9. Scheduling / Resource Allocation
10. Replace web2py helpers with dojo.dijits

What makes up a plug-in?

A plug-in can define any of the following structures.

1. Database models
2. Functions for data model integration
3. Controllers
4. Views
5. Modules

File Layout of a Plug-In

Plug-ins are application specific.

Plug-in files will be located alongside your application files. Any file belonging to a plug-in will follow a naming convention where `plugin_<name>.py`.

Refer to the following tree as an example.

- welcome
 - controllers
 - plugin_comments.py
 - models
 - plugin_comments.py
 - modules
 - plugin_comments.py OR
 - plugin_comments
 - moduleA.py
 - moduleB.py
 - views
 - plugin_comments
 - index.html
 - view.html
 - post.html
 - static
 - plugin_comments
 - css
 - js
 - img
 - plugins
 - plugin_comments.readme
 - plugin_comments.license
 - plugin_comments.meta

For models, you can alternatively place plug-in files in a sub-folder approach.

****This is not yet supported fully***

- welcome
 - models
 - plugin_comments
 - init.py
 - commenting.py
 - cleanup.py

The PluginManager Extension

To use plug-ins you will need to make use of the web2py PluginManager class. The purpose of the PluginManager is to manage all aspects of plug-ins. The control ranges from the access to objects in a local namespace. Exposing public functions and queries. The PluginManager acts like a Storage dictionary.

Declaring the PluginManager

You must instantiate the PluginManager class before any plug-ins are executed. Plug-ins will get their DAL object from the PluginManager, so it is necessary to declare this. This is so that plug-ins will not be accessing the global db object which might contain business data.

PluginManager takes one argument, a DAL object, if you do not manually specify a db object for a plug-in, it will assume PluginManagers db instance. Once you have defined your PluginManager instance, you can set any of the plug-ins configurable options, including db.

```
## PluginManager Declaration (db.py)
db = DAL('sqlite://business_data.sqlite')
pdb = DAL('sqlite://plugin_data.sqlite')
commentsdb = DAL('sqlite://comments.sqlite')
```

```

"""
plugins = PluginManager(pdb)
plugins.comments.db = commentsdb
plugins.comments.linked_to = db.post
plugins.comments.require_captcha = False
plugins.ratings.num_stars = 5

```

Implementation of PluginManager

PluginManager will need to be smarter than a standard Storage object. Below is an example implementation of the PluginManager class.

```

def __init__(self, db, **kwargs):
    self._plugin_db = db
    if not db:
        raise Exception("You must define a default database for plug-ins.")

    dict.__init__(self, **kwargs)

def __getattr__(self, key):
    try:
        return self[key]
    except KeyError:
        return self[key] = Storage(
            db = self._plugin_db,
            qry = Storage(),
        )

```

Data Models

A plug-in will be able to define database/data models. These data models can reference and link to existing data models. A plug-in database models should not be accessed directly by the parent application, yet it is possible. Plug-ins should provide queries and functions for accessing the data models.

Plug-in models will be executed in a local environment, they are not allowed to pollute the global namespace of a web2py application. All functions and queries will be exposed publicly via the PluginManager instance. To execute a plug-ins models in a local environment, it will be wrapped in a generic function. This provides that plug-ins stay local, and to shorten access to that plug-ins settings and variables.

If a plug-in defines queries, it should not perform `.select` on the data set, this is so that the programmer using the plug-in can define their own cache, limitby, and orderby settings.

Plug-in queries will be accessible by the `plugins.<name>.qry` storage object. Queries can be any function that expects arguments to build the query, or a `gluon.sql.Set` object.

Plug-ins “may” expose functions and methods into the global name space if absolutely necessary. This is mainly for situations where the plug-in is overwriting base web2py objects. For example, a plug-in that redefines all of the web2py helper objects to use `dojo.dijits` instead of `html`.

Plug-ins should NOT access any variables that are located in the global namespace such as `'db'`. They should always access these types of objects through the PluginManager. This way the programmer has control over what the plug-in has access to. This is not a security measure however, it is just a convention and it is still up to the developer to check a plug-ins code before installing it!

Plug-ins can provide default values for options if they do not exist in the object. Plug-ins should not overwrite any settings that have been defined before the plug-in.

```

## Data Model Declaration (plugin_comments.py)
def _(p):

```

```

p.db.define_table('plugin_comments',
    Field('name'), Field('content'),
    Field('reference_table', p.linked_to)
)

def get_comments_for(fkid):
    return p.db(p.db.plugin_comments.reference_table == fkid)
p.qry.get_comments_for = get_comments_for
p.qry.get_latest_comments = p.db(p.db.plugin_comments.id > 0)

# Now we call our local function with the instance of our plug-in.
_(plugins.comments)

```

For everything else theres...

For controllers, views, modules, static files, work exactly like any other web2py object. A plug-in controller will use the plugins.<name> object to work with the data. A controller can be called by LOAD or as a normal web2py routing. The only specification here is that plug-in file names follow the format as specified in the File Layout section.

Plugin URL Mapping

Plug-ins should have a way to expose controller functions that get mapped by URL. URLs can be mapped with the existing web2py system.

```

URL(r=request, c='plugin_comments', f='post', args=[183])

/welcome/plugin_comments/post/183

```

Plugin Session Handling

Certain plug-ins will need the ability to store variables in the session. Plug-ins may access the session object, as long as they follow a naming convention of:

```

session.<plugin_name> = Storage()
session.<plugin_name>.a_session_var = "hi"

if not session.comments:
    session.comments = Storage()
session.comments.last_comment_id = 183

```

Plugin Crontab

The current crontab system will need to be patched to allow cron in multiple files. This way in the cron folder there could be a file called "plugin_<name>.crontab" which could be easily deleted with the plug-in packing and removal system.

Management and Installation/Removal

Plug-ins include information regarding version, author, licensing and other meta information about a plug-in. There will be three files for every plug-in, located in plugins folder of the application. Following the naming convention, the three files are:

- plugin_<name>.readme
- plugin_<name>.license
- plugin_<name>.meta

Readme Information

plugin_<name>.readme will contain instructions for usage of the plug-in, nuances of the plug-in, and other information a developer might need to know about the plug-in.

License Information

plugin_<name>.license will contain the full license text that the plug-in is distributed under.

Meta Information

plugin_<name>.meta will contain the following information. This will be in a format that the web2py admin application can autodiscover, a python dict, and determine a plug-ins information.

- Plugin Name (ex: wComments)
- Author
- Author Contact
- Version
- Short License (ex: GPL v3)

A plug-ins version scheme should follow a Gnu style versioning format. [1] This format denotes as Major_Version_Number.Minor_Version_Number [. Revision_Number [. Build_Number]]
Example: 1.2.1, 2.0, 5.0.0 build-13124

This is so that admin can easily update a plug-in based on version.

Below is an example plugin_comments.meta file. It is a python dict format.

```
plugin_comments_meta: {
    'plugin_name': 'wComments',
    'author': 'Joe Smith',
    'author_contact': 'joe@smith.com',
    'version': '1.3.5',
    'license': 'GPL v3',
}
```

Installation / Removal

The admin interface will handle the installation of a plug-in by unpacking an archive file. When plugincentral.web2py.com is completed, installation/updating could be automated by providing the plug-in URL or name.

More

This paper was a look into simple plug-in systems such as commenting and ratings. However a true web2py plug-in system would include more than just a framework for data. Plug-ins could integrate into web2py itself, overwriting default web2py functionality, look and feel, like a layout plug-in. A plug-in could overload the default web2py helpers with dojo.dijit widgets for example.

Things that are still left to do:

1. Refine plugin conventions
2. Implement PluginManager
3. Patch cron for multi-file access
4. Patch admin to discover plug-in readme, license, and meta.
5. Finish plugincentral

Things still to discuss:

[1] <http://www.codeweblog.com/software-version-number-and-format-of-the-naming-rules/>

1. Affirm convention for plug-in storing session variables
2. Mechanism of hooks for plug-ins to register stuff with the app (such as a change menu)
3. How plug-in dependencies could be handled ²
4. Web2py level plug-ins vs. application level plug-ins. ³

² <https://mail.google.com/mail/?shva=1#search/label%3Aweb2py+plugin/1275053202d763e5>

³ <https://mail.google.com/mail/?shva=1#search/label%3Aweb2py+plugin/1275053202d763e5>