Flask-Plugins Documentation

Release 0.1-dev

sh4nks

January 26, 2016

Contents

Flask-Plugins provides an easy way to create plugins for your application. It is possible to create Events which can then be used to extend your application without the need to modify your core code.

CHAPTER 1

Quickstart

First of all, you have to initialize the extension. This can be done in two different ones.

The first one is to initalize it directly:

```
from flask.ext.plugins import PluginManager

plugin_manager = PluginManager(app)
```

where as the second one is to use the factory pattern:

```
from flask.ext.plugins import PluginManager

plugin_manager = PluginManager()
plugin_manager.init_app(app)
```

1.1 Plugin Structure

After the first step is done, you can start developing your first plugin. The most minimal plugin needs to have at least it's **own** directory, a **info.json** file, where some meta data about the plugin is stored and last but not least a **__init__.py** file where the name of the plugin class is specified.

For example, the structure of small plugin can look like this:

```
my_plugin
|-- info.json
|-- __init__.py
```

the structure of a more complex plugin can also look like this:

```
my_plugin
|-- info.json               Contains the Plugin's metadata
|-- license.txt             The full license text of your plugin
|-- __init__.py             The plugin's main class is located here
|-- views.py
|-- models.py
|-- forms.py
|-- static
|   |-- style.css
|-- templates
    |-- myplugin.html
```

1.2 Hello World!

For a better understanding you can also have a look at the example application.

Another important note is, that you have to specify the name of the plugin class in the **__init__.py** file. The reason for this is that the Plugin Loader looks in the `__init__.py` for a `__plugin__` variable to load the plugin. If no such variable exists, the loader will just go on to the next plugin. and if the specified name in `__init__.py` doesn't match the name of the actual plugin class it will raise an exception.

So for example, the `__plugin__` variable, in the `__init__.py` file, for a `HelloWorld` plugin class could look like this:

```
__plugin__ = "HelloWorld"
```

A HelloWorld Plugin could, for example, look like this:

```
class HelloWorld(Plugin):
    def setup(self):
        connect_event('before-data-rendered', do_before_data_rendered)
```

In addition to this, the **info.json** file is also required. It just contains some information about the plugin:

```
{
    "identifier": "hello_world",
    "name": "Hello World",
    "author": "sh4nks",
    "license": "BSD",
    "description": "A Hello World Plugin.",
    "version": "1.0.0"
}
```

For more available fields, see *The info.json File*.

1.3 Enabling and Disabling Plugins

This extension, unlike other python plugin systems, uses a different approach for handling plugins. Instead of installing plugins from PyPI, plugins should just be dropped into a directory. Another thing that is unique, is to disable plugins without touching any source code. To do so, a simple *DISABLED* file in the plugin's root directory is enough. This can either be done by hand or with the methods provided by *PluginManager*.

The directory structure of a disabled plugin is shown below.

```
my_plugin
|-- DISABLED            # Just add a empty file named "DISABLED" to disable a plugin
|-- info.json
|-- __init__.py
```

The server needs to be restarted in order to disable the plugin. This is a limitation of Flask. However, it is possible, to restart the application by sending a HUP signal to the application server. The following code snippets, are showing how this can be done with the WSGI server *gunicorn*. Gunicorn has be to started in daemon (`--daemon`) mode in order for this to work.

```
@app.route('/restart-server/')
def restart_server():
  os.kill(os.getpid(), signal.SIGHUP)
```

Which you can then call via a AJAX call.

```
function reload_server() {
  // Reload Server
  $.ajax({
    url: "/reload-server/"
  });
  // Wait 1 second and reload page
  setTimeout(function(){
    window.location = document.URL;
  }, 1000);
}
```

This can then be called with a simple button (given you have included the JS file in your html template).

```
<button onclick='reload_server()'>Reload Server</button>
```

1.4 Events

We also provide a Event system out of the box. It is up to you if you want to extend your application with events. If you decide to use it, then you just need to add in specific places in your code the *emit_event()* function with the name of your event and optionally the data which can be modified by a plugin:

```
from flask.ext.plugins import emit_event

emit_event("before-data-rendered", data)
```

and than you can add a callback (e.g. in your plugin setup method):

```
from flask.ext.plugins import connect_event

def do_before_data_rendered(data):
    return "returning modified data"

connect_event("before-data-rendered", do_before_data_rendered)
```

Of course you can also do that in your templates - For this we have already added *emit_event()* to your jinja env context. So you just need to call it in the template:

```
{{ emit_event("before-data-rendered") }}
```

If you want to see a fully working example, please check it out here.

CHAPTER 2

The info.json File

Below are shown all available fields a plugin can use. Of course, it always depends if the application, that uses this extension, needs so much information about a plugin. The only really required fields are marked with **required**.

`identifier`: required The plugin's identifier. It should be a Python identifier (starts with a letter or underscore, the rest can be letters, underscores, or numbers) and should match the name of the plugin's folder.

`name`: required A human-readable name for the plugin.

`author`: required The name of the plugin's author, that is, you. It does not have to include an e-mail address, and should be displayed verbatim.

`description` A description of the plugin in a few sentences. If you can write multiple languages, you can include additional fields in the form `description_lc`, where `lc` is a two-letter language code like `es` or `de`. They should contain the description, but in the indicated language.

`description_lc` This is a dictionary of localized versions of the description. The language codes are all lowercase, and the `en` key is preloaded with the base description.

`website` The URL of the plugin's Web site. This can be a Web site specifically for this plugin, Web site for a collection of plugins that includes this plugin, or just the author's Web site.

`license` A simple phrase indicating your plugin's license, like `GPL`, `MIT/X11`, `Public Domain`, or `Creative Commons BY-SA 3.0`. You can put the full license's text in the `license.txt` file.

`license_url` A URL pointing to the license text online.

`version` This is simply to make it easier to distinguish between what version of your plugin people are using. It's up to the theme/layout to decide whether or not to show this, though.

`options` Any additional options. These are entirely application-specific, and may determine other aspects of the application's behavior.

CHAPTER 3

API Documentation

flask_plugins.**get_enabled_plugins**()
Returns all enabled plugins as a list

flask_plugins.**get_all_plugins**()
Returns all plugins as a list including the disabled ones.

flask_plugins.**get_plugin_from_all**(*identifier*)
Returns a plugin instance from all plugins (includes also the disabled ones) for the given name.

flask_plugins.**get_plugin**(*identifier*)
Returns a plugin instance from the enabled plugins for the given name.

3.1 The Plugin Class

Every `Plugin` should implement this class. It is used to get plugin specific data. and the *`PluginManager`* tries call the methods which are stated below.

class flask_plugins.**Plugin**(*path*)
Every plugin should implement this class. It handles the registration for the plugin hooks, creates or modifies additional relations or registers plugin specific thinks

enabled = False
If setup is called, this will be set to `True`.

path = None
The plugin's root path. All the files in the plugin are under this path.

name = None
The plugin's name, as given in info.json. This is the human readable name.

identifier = None
The plugin's identifier. This is an actual Python identifier, and in most situations should match the name of the directory the plugin is in.

description = None
The human readable description. This is the default (English) version.

description_lc = None
This is a dictionary of localized versions of the description. The language codes are all lowercase, and the `en` key is preloaded with the base description.

author = None
The author's name, as given in info.json. This may or may not include their email, so it's best just to display it as-is.

license = None
A short phrase describing the license, like "GPL", "BSD", "Public Domain", or "Creative Commons BY-SA 3.0".

license_url = None
A URL pointing to the license text online.

website = None
The URL to the plugin's or author's Web site.

version = None
The plugin's version string.

options = None
Any additional options. These are entirely application-specific, and may determine other aspects of the application's behavior.

license_text
The contents of the theme's license.txt file, if it exists. This is used to display the full license text if necessary. (It is *None* if there was not a license.txt.)

setup()
This method is used to register all things that the plugin wants to register.

enable()
Enables the plugin by removing the 'DISABLED' file in the plugins root directory, calls the `setup()` method and sets the plugin state to true.

disable()
Disablesthe plugin.

The app usually has to be restarted after this action because plugins _can_ register blueprints and in order to "unregister" them, the application object has to be destroyed. This is a limitation of Flask and if you want to know more about this visit this link: http://flask.pocoo.org/docs/0.10/blueprints/

install()
Installs the things that must be installed in order to have a fully and correctly working plugin. For example, something that needs to be installed can be a relation and/or modify a existing relation.

uninstall()
Uninstalls all the things which were previously installed by *install()*. A Plugin must override this method.

3.2 Plugin System

class flask_plugins.**PluginManager**(*app=None*, ***kwargs*)
Collects all Plugins and maps the metadata to the plugin

__init__(*app=None*, ***kwargs*)
Initializes the PluginManager. It is also possible to initialize the PluginManager via a factory. For example:

```
plugin_manager = PluginManager()
plugin_manager.init_app(app)
```

Parameters

- **app** – The flask application.
- **plugin_folder** – The plugin folder where the plugins resides.
- **base_app_folder** – The base folder for the application. It is used to build the plugins package name.

all_plugins
Returns all plugins including disabled ones.

plugins
Returns all enabled plugins as a dictionary. You still need to call the setup method to fully enable them.

load_plugins()
Loads all plugins. They are still disabled. Returns a list with all loaded plugins. They should now be accessible via self.plugins.

find_plugins()
Find all possible plugins in the plugin folder.

setup_plugins()
Runs the setup for all enabled plugins. Should be run after the PluginManager has been initialized. Sets the state of the plugin to enabled.

install_plugins(*plugins=None*)
Installs one or more plugins.

Parameters **plugins** – An iterable with plugins. If no plugins are passed it will try to install all plugins.

uninstall_plugins(*plugins=None*)
Uninstalls one or more plugins.

Parameters **plugins** – An iterable with plugins. If no plugins are passed it will try to uninstall all plugins.

enable_plugins(*plugins=None*)
Enables one or more plugins.

It either returns the amount of enabled plugins or raises an exception caused by `os.remove` which says most likely that you can't write on the filesystem.

Parameters **plugins** – An iterable with plugins.

disable_plugins(*plugins=None*)
Disables one or more plugins. It either returns the amount of disabled plugins or raises an exception caused by `open` which says most likely that you can't write on the filesystem.

The app usually has to be restarted after this action because plugins **can** register blueprints and in order to "unregister" them, the application object has to be destroyed. This is a limitation of Flask and if you want to know more about this visit this link: http://flask.pocoo.org/docs/0.10/blueprints/

Parameters **plugins** – An iterable with plugins

3.3 Event System

class flask_plugins.**EventManager**(*app*)
Helper class that handles event listeners and event emitting.

This is *not* a public interface. Always use the *emit_event* or *connect_event* or the *iter_listeners* functions to access it.

connect(*event*, *callback*, *position='after'*)
Connect a callback to an event.

remove(*event*, *callback*)
Remove a callback again.

iter(*event*)
Return an iterator for all listeners of a given name.

template_emit(*event*, **args*, ***kwargs*)
Emits events for the template context.

flask_plugins.**emit_event**(*event*, **args*, ***kwargs*)
Emit a event and return a list of event results. Each called function contributes one item to the returned list.

This is equivalent to the following call to *iter_listeners()*:

```
result = []
for listener in iter_listeners(event):
    result.append(listener(*args, **kwargs))
```

flask_plugins.**connect_event**(*event*, *callback*, *position='after'*)
Connect a callback to an event. Per default the callback is appended to the end of the handlers but handlers can ask for a higher privilege by setting *position* to `'before'`.

Example usage:

```
def on_before_metadata_assembled(metadata):
    metadata.append('<!-- IM IN UR METADATA -->')

# And in your setup() method do this:
    connect_event('before-metadata-assembled',
                  on_before_metadata_assembled)
```

flask_plugins.**iter_listeners**(*event*)
Return an iterator for all the listeners for the event provided.

Symbols

A

C

D

E

F

G

I

L

N

O

P

R

S

T

U

V

W