-plugins Documentation

Release 0.2.1

Mar 31, 2017

Contents

CHAPTER 1

Introduction

`django-plugins` will help you to make your Django app more reusable. You will be able to define plugin points, plugins and various ways, how plugins can be integrated to your base app and extended from other apps providing plugins.

The idea for `django-plugins` was taken from Marty Alchin blog, for in deep understanding about how this plugin system work, read Marty Alchin blog.

Features

- Synchronization with database.
- Plugin management from Django admin.
- **Model fields:**
  - `PluginField`
  - `ManyPluginField`
- **Form fields:**
  - `PluginChoiceField`
  - `PluginModelChoiceField`
  - `PluginMultipleChoiceField`
  - `PluginModelMultipleChoiceField`
- Possibility to include plugins to urls.
- Possibility to access plugins from templates.
- Many ways to access plugins and associated models.

CHAPTER 2

Use case

`django-plugins` can be used in those situations where instances of your particular model can behave differently.

For example, you have one `Node` model:

```
class Node(models.Model):
    title = models.CharField(max_length=255)
    body = models.TextField()
```

This model stores basic information for news, articles and other possible content types. Each different content type has different forms, different templates for displaying and listing content.

To implement all this, you simply can use `django-plugins`:

```
class Node(models.Model):
    title = models.CharField(max_length=255)
    body = models.TextField()
    content_type = PluginField(ContentType)
```

Then in your `views.py` you do:

```
@render_to('create.html')
def node_create(request, plugin):
    return {'form': plugin.get_form()}

@render_to('update.html')
def node_update(request, plugin, node_id):
    node = get_object_or_404(Node, pk=node_id)
    return {'form': plugin.get_form(instance=node)}

@render_to()
def node_read(request, node_id):
    node = get_object_or_404(Node, pk=node_id)
    plugin = node.content_type.get_plugin()
    return {
        'TEMPLATE': plugin.get_template(),
```

```
        'plugin': plugin,
        'node': node,
    }
```

CHAPTER 3

How to use it in your app?

All plugin points and plugins live in `plugins.py` file in your django app folder.

Example how to register a plugin point:

```
from djangoplugins.point import PluginPoint

class MyPluginPoint(PluginPoint):
    """
    Documentation, that describes how plugins can implement this plugin
    point.

    """
    pass
```

Example, how to register plugin, that implements `MyPluginPoint`, defined above:

```
class MyPlugin1(MyPluginPoint):
    name = 'plugin-1'
    title = 'Plugin 1'

class MyPlugin2(MyPluginPoint):
    name = 'plugin-1'
    title = 'Plugin 2'
```

All plugins must define at least `name` and `title` attributes. These properties are used everywhere in plugin system.

`name` This is a slug like name, used in urls and similar places.

`title` Any human readable title for plugin. Value of this attribute will be shown to users everywhere.

CHAPTER 4

Database

All defined plugins and plugin points are synchronized to database using Django management command `syncplugins` or `syncdb`. `syncdb` should be always enough, but some times, if you added or changed plugins code and need to update those changes to database, but don't want anything more, then you should use `syncplugins` management command.

When added to database, plugins can be ordered, disabled, accessed from Django admin, etc.

`syncplugins` command detects if plugins or plugin points where removed from code and marks them as `REMOVED`, but leaves them in place. If you want to clean up your database and really delete all removed plugins us `--delete` flag.

CHAPTER 5

Utilizing available plugins

There are many ways how you can use plugins and plugin points. Out of the box plugins are stored as python objects and synchronized to database called plugin models.

Each plugin is linked to one record of `djangoplugins.models.Plugin` model. Plugins provides all login, plugin models provides all database possibilities, like sorting, searching, filtering. Combining both we get powerful plugin system.

Plugin classes are hardcoded and cannot be modified by users directly. But users can modify database instances linked to those hardcoded plugins. Thats why you should always trust database instances, but no hardcoded plugins, because users can change some thing in database and expects to see those changes in his web site.

Plugin and plugin models, both has `name` and `title` attributes, but you should always use these attributes from model instances, but not from plugins.

Here is example to illustrate this:

```
BAD:

plugin = MyPlugin()
print(plugin.title)

GOOD:

plugin = MyPlugin()
if plugin.is_active():
    print(plugin.get_model().title)
```

As you see, in GOOD example, we also check if a plugin is active. Users can enable or disable plugins using admin. Thats why you should always check if a plugin is active, before using it. Using methods like `get_plugins` and `get_plugins_qs` you will always get only active plugins. So checking if plugin is active is needed only if you working with particular plugin, bet not with all plugins of a point.

`get_plugins` method of each plugin point class and plugin point model instance, returns list of all active plugin instances.

Example, how to use it:

```
from my_app.plugins import MyPluginPoint

@register.inclusion_tag('templatetags/actions.html', takes_context=True)
def my_plugins(context):
    plugins = MyPluginPoint.get_plugins()
    return {'plugins': plugins}
```

templatetags/actions.html:

```
<ul>
    {% for plugin in plugins %}
    <li>plugin.title</li>
    {% endfor %}
</ul>
```

If you need to sort or filter plugins, you should always access them via Django ORM:

```
from my_app.plugins import MyPluginPoint

@render_to('my_app/my_template.html')
def my_view(request):
    return {
        'plugins': MyPluginPoint.get_plugins_qs().order_by('name')
    }
```

CHAPTER 6

Model fields

You can tie your models with plugins. Using example below, plugins can be assigned to model instances:

```
from django.db import models
from djangoplugins.fields import PluginField
from my_app.plugins import MyPluginPoint

class MyModel(models.Model):
    plugin = PluginField(MyPluginPoint)
```

Also there is `ManyPluginField`, for many-to-many relation.

PluginField

class `PluginField` (*point*[, ***options*])

This field is simply foreign key to `Plugin` model.

Takes one extra required argument:

`ForeignKey`**`.point`**
Plugin point class.

ManyPluginField

class `ManyPluginField` (*point*[, ***options*])

Takes one extra required argument, `point`, as for `PluginField`.

CHAPTER 7

Form fields

It's easy to put your plugin point to forms using set of plugin fields for forms:

```
from django import forms
from djangoplugins.fields import (
        PluginChoiceField, PluginMultipleChoiceField,
        PluginModelChoiceField, PluginModelMultipleChoiceField,
    )
from my_app.plugins import MyPluginPoint

class MyForm(forms.Form):
    # Two fields below provides simple ChoiceField with choices of plugins.
    choice = PluginChoiceField(MyPluginPoint)
    # This field currently disabled:
    #   http://code.djangoproject.com/ticket/9161
    #multiple_choice = PluginMultipleChoiceField(MyPluginPoint)

    # These two fields below provides ModelChoiceField with queryset of
    # plugis.
    model_choice = PluginModelChoiceField(MyPluginPoint)
    model_multiple_choice = PluginModelMultipleChoiceField(MyPluginPoint)
```

PluginChoiceField

class `PluginChoiceField`(***kwargs*)

- Default widget: `Select`
- Empty value: `''` (an empty string)
- Normalizes to: Plugin object.
- Validates that the given value is valid plugin name of specified plugin point.
- Error message keys: `required`, `invalid_choice`

This field can be used, when you want to validate if a string is valid plugin name and that plugin belongs to specified plugin point.

Also this field normalizes to plugin object instance, but not to plugin model instance.

Takes one extra required argument:

`PluginChoiceField.`**`point`**
Plugin point class.

PluginMultipleChoiceField

Note: Currently this field is disabled due bug in Django:

http://code.djangoproject.com/ticket/9161

class `PluginMultipleChoiceField`(***kwargs*)

- Default widget: `SelectMultiple`
- Empty value: `[]` (an empty list)
- Normalizes to: A list of Plugin objects.
- Validates that every value in the given list of values is valid plugin name of specified plugin point.
- Error message keys: `required`, `invalid_choice`, `invalid_list`

Takes one extra required argument, `point`, as for `PluginChoiceField`.

PluginModelChoiceField

class `PluginModelChoiceField`(***kwargs*)

- Default widget: `Select`
- Empty value: `None`
- Normalizes to: A Plugin model instance.
- Validates that the given id is plugin id of specified plugin point.
- Error message keys: `required`, `invalid_choice`

Takes one extra required argument, `point`, as for `PluginChoiceField`.

PluginModelMultipleChoiceField

class `PluginModelMultipleChoiceField`(***kwargs*)

- Default widget: `SelectMultiple`
- Empty value: `[]` (an empty list)
- Normalizes to: A list of Plugin model instances.
- Validates that every id in the given list of values is plugin id of specified plugin point.

- Error message keys: `required`, `list`, `invalid_choice`, `invalid_pk_value`

Takes one extra required argument, `point`, as for `PluginChoiceField`.

CHAPTER 8

Urls

`django-plugins` has build-in possibility to include urls from plugins. Here is example how this can be done:

```
from django.conf.urls.defaults import patterns
from plugins.utils import include_plugins
from my_app.plugin_points import MyPluginPoint

urlpatterns = patterns('wora.views',
    (r'^plugin/', include_plugins(MyPluginPoint)),
)
```

`include_plugins` function will search `get_urls` and `name` attributes in all plugins, and if both are available, then provided urls will be included.

Example plugin:

```
class MyPluginWithUrls(MyPluginPoint):
    name = 'my-plugin'
    title = 'My plugin'

    def get_urls(self):
        return patterns('my_app.views',
                url(r'create/$', 'create', name='my-app-create'),
                url(r'read/$', 'read', name='my-app-read'),
                url(r'update/$', 'update', name='my-app-update'),
                url(r'delete/$', 'delete', name='my-app-delete'),
            )
```

With this plugin, plugin point inclusion will provide these urls:

```
/plugin/my-plugin/create/
/plugin/my-plugin/read/
/plugin/my-plugin/update/
/plugin/my-plugin/delete/
```

Plugin points are better place to define urls. Here is example, how all this can be done:

```
class MyPluginPoint(PluginPoint):
    def get_urls(self):
        return patterns('my_app.views',
                url(r'create/$', 'create',
                    name='my-app-%s-create' % self.name),
            )

class MyPlugin1(MyPluginPoint):
    name = 'my-plugin-1'
    title = 'My Plugin 1'

class MyPlugin2(MyPluginPoint):
    name = 'my-plugin-2'
    title = 'My Plugin 2'

class MyPlugin3(MyPluginPoint):
    name = 'my-plugin-3'
    title = 'My Plugin 3'
```

From all these plugins, these urls will be available:

```
/plugin/my-plugin-1/create/
/plugin/my-plugin-2/create/
/plugin/my-plugin-3/create/
```

In templates all these urls can be added using these url names:

```
{% url my-app-my-plugin-1-create %}
{% url my-app-my-plugin-2-create %}
{% url my-app-my-plugin-3-create %}
```

CHAPTER 9

Templates

You can access your plugins in templates using `get_plugins` template tag.:

```
{% load plugins %}
{% get_plugins my_app.plugins.MyPluginPoint as plugins %}
<ul>
    {% for plugin in plugins %}
    <li>{{ plugin.title }} {{ plugin.get_plugin.plugin_class_attr }}</li>
    {% endfor %}
</ul>
```

In example above, `get_plugins` returns ordered queryset of plugin models, but not plugins directly.

CHAPTER 10

Using plugins with Django ORM

It is possible to use plugins with Django ORM.

If your model has plugin field, you can:

```
from my_app.models import MyModel
from my_app.plugins import MyPlugin

plugin_model = MyPlugin.get_model()

qs = MyModel.objects.\
        filter(name='name', plugin=plugin_model).\
        order_by('plugin__order')

qs = MyModel.objects.filter(plugin__name='email')
```

As mentioned above, you can get queryset of all plugins from a plugin point easily:

```
count = MyPluginPoint.get_plugins_qs().count()
```

CHAPTER 11

How to get all plugins?

There are two ways, how you can get all plugins of a plugin point:

```
MyPluginPoint.get_plugins()
```

and:

```
MyPluginPoint.get_plugins_qs()
```

First example returns plugins directly in random order. Second example returns Django queryset with plugin models ordered by `order` field.

CHAPTER 12

How to get model instance of a plugin?

In example below are listed all possible ways, how you can get model instance of a plugin.

```
plugin = MyPlugin()

# Get model instance from plugin instance.
plugin_model = plugin.get_model()

# Get model instance from plugin class.
plugin_model = MyPlugin.get_model()

# Get model instance by plugin name.
plugin_model = MyPluginPoint.get_model('my-plugin')

# Get model instance of a plugin point:
plugin_point_model = MyPluginPoint.get_model()
```

`get_model` method can raise `ObjectDoesNotExist` exception, so you should check it:

```
try:
    plugin_model = MyPlugin.get_model()
except MyPlugin.DoesNotExist:
    plugin_model = None
```

CHAPTER 13

How to get plugin from a model instance?

Easy:

```
plugin = plugin_model.get_plugin()
```

CHAPTER 14

Why another plugin system?

Currently these similar projects exists:

- django-app-plugins - template oriented, pretty complete, but totally undocumented. Project is not active and bugs are fixed only in forked repository django-caching-app-plugins.
- django-plugins - template oriented, small project. Plugins are uploaded through Django admin.

Also there is a lot of articles and code snippets, that describes how plugin system can be implemented. Here is article, that most influenced this project:

- http://martyalchin.com/2008/jan/10/simple-plugin-framework/

Also see list of other articles and python plugin system implementations:

- http://wehart.blogspot.com/2009/01/python-plugin-frameworks.html

None of these projects fully provides what I need:

- Good documentation.
- Plugins and plugin points should be provided by Django apps, not only by single uploaded files.
- Plugins should not be restricted by file names, then can be registered anywhere, like Django signals.
- Plugins should be synchronized with database, and plugin point can be used as fields.

Index