django-textplusstuff Documentation

Release 0.1

Jonathan Ellenberger et al

Feb 22, 2018

Contents

CHAPTER 1

About

1.1 Summary

A django field that makes it easy to intersperse 'stuff' into blocks of text.

1.2 Documentation

Full documentation available at Read the Docs.

1.2.1 A Flexible Interface

`django-textplusstuff` provides a simple interface for returning the contents of your field however you like: as either markdown-flavored text, valid HTML markup (with or without 'stuff' interspersed) or even plain text (with all markdown formatting removed).

1.2.2 Keep Track of Your Content

`django-textplusstuff` also keeps track of which model instances are associated within each TextPlusStuffField (via the *TextPlusStuffLink* model) so you can see where all your textplusstuff-integrated content is used across your django project.

1.2.3 Easy Integration

Registering existing models for use in TextPlusStuffFields is as easy as integrating a model into the admin.

1.2.4 Designer/Front-End Developer Friendly

Each model registered with `django-textplusstuff` can have as many 'renditions' as you like which keeps business logic DRY while enabling designers and front-end developers to have control over how content is displayed.

1.3 Current Version

0.6

1.4 Dependencies

- `markdown2` >= 2.3.x
- `beautifulsoup4` >= 4.4.0
- `django` >= 1.6.x
- `djangorestframework` >= 2.4.4

1.4.1 Python Compatibility

- 2.7.x
- 3.4.x
- 3.5.x

1.4.2 Django Compatibility

- 1.7.x
- 1.8.x
- 1.9.x

1.4.3 Django REST Framework Compatibility

- 2.4.4
- 3.0.x
- 3.1.x
- 3.2.x
- 3.3.x (**NOTE:** Django 1.6.x is not compatible with DRF 3.3.x)

CHAPTER 2

Contents

2.1 Installation Instructions

Installation is easy with `pip`:

1. Installation is easy with pip:

```
$ pip install django-textplusstuff
```

> **Note:** `django-textplusstuff` will not install `django`.

2. Add required settings:

   Add *textplusstuff* to *INSTALLED_APPS*:

```
INSTALLED_APPS = (
    # Other apps here
    'rest_framework',
    'textplusstuff',
)
```

   Add the TEXTPLUSSTUFF_STUFFGROUPS setting with at least one StuffGroup. It can be named whatever you like (the one below is just 'example'):

```
TEXTPLUSSTUFF_STUFFGROUPS = {
    'example': {
        'name': 'Example',
        'description': "This is an example of a StuffGroup!"
    },
}
```

Note: StuffGroups are used to organize Stuff in the upcoming editor tool and are required when you register Stuff.

3. Add textplusstuff-required bits to your project's base *urls.py*:

```
# Base project urls.py
from django.conf.urls import patterns, include, url
from django.contrib import admin

# Importing required textplusstuff bits
from textplusstuff.registry import stuff_registry

urlpatterns = patterns(
    '',
    # Admin URLs
    url(r'^admin/', include(admin.site.urls)),
    # textplusstuff URLs
    url(r'^textplusstuff/', include(stuff_registry.urls))
)
```

2.2 Using textplusstuff

2.2.1 Registering Stuff

To start using textplusstuff you have to register a model as Stuff. The examples below will use the creatively named *TestModel* which has one attribute, 'name' a *CharField*:

1. Create a file called *serializers.py* within the app that has the model you want to register as stuff:

```
someproject/
    someapp/
        models.py
        serializers.py # Like this!
```

2. Now open *serializers.py* to create your first serializer. For more information on serializing models check out django REST frameworks fantastic docs.:

```
# serializers.py

from rest_framework.serializers import ModelSerializer

from .models import TestModel

class TestModelSerializer(ModelSerializer):

    class Meta:
        model = TestModel
        fields = (
            'name',
        )
```

3. OK, now that we've got a serializer when need to create a file called `stuff.py` within the app that has the model you want to register as Stuff:

```
someproject/
    someapp/
        models.py
        serializers.py
        stuff.py # Like this!
```

4. Now open the *stuff.py* file you just created and import the model you want to register and the serializer you just created:

```
# someapp/stuff.py
from textplusstuff import registry

from .models import TestModel
from .serializers import TestModelSerializer

class TestModelStuff(registry.ModelStuff):
    # The queryset used to retrieve instances of TestModel
    # within the front-end interface. For instance, you could
    # exclude 'unpublished' instances or anything else you can
    # query the ORM against
    queryset = TestModel.objects.all()

    # What humans see when they see this stuff
    verbose_name = 'Test Model'
    verbose_name_plural = 'Test Models'
    description = 'Add a Test Model'

    # The serializer we just defined, this is what provides the context/
↪JSON
    # payload for this Stuff
    serializer_class = TestModelSerializer

    # All Stuff must have at least one rendition (specified in
    # the `renditions` attribute below) which basically
    # just points to a template and some human-readable metadata.
    # At present there are only two options for setting rendition_type:
    # either 'block' (the default) or inline. These will be used by
    # the front-end editor when placing tokens.
    renditions = [
        registry.Rendition(
            short_name='sidebar_left',
            verbose_name='Test Model Sidebar',
            description='Displays a Test Model in the sidebar.',
            path_to_template='someapp/templates/sidebar_left.html',
            rendition_type='block'
        )
    ]
    # The attributes used in the list (table) display of the front-end
    # editing tool.
    list_display = ('id', 'name')

# OK, now let's register our Model and its Stuff config:
registry.stuff_registry.add_modelstuff(
    TestModel,
    TestModelStuff,
    groups=['image', 'media']
)
```

Once you've registered your Stuff you can test if it worked by firing up a webserver and visiting http://localhost:8000/textplusstuff/.

2.2.2 Registering 'Non-Core' Renditions

Sometimes you'll want to add an additional rendition to some Stuff registered in a separate third-party application. Previously you'd have to do a bunch of boilerplate to accomplish this (unregister the model in question, import both it and its Stuff configuration, subclass the Stuff config, modify the `renditions` attribute and then re-register the subclassed Stuff config with `textplusstuff.registry.stuff_registry`). The `0.3` release introduced a painless way to register 'non-core' renditions with an already registered Stuff class.

Here's how to do it with our *TestModel* example:

```
# anotherapp/stuff.py

from textplusstuff import registry

from someapp.models import TestModel

registry.stuff_registry.add_noncore_modelstuff_rendition(
    TestModel,
    registry.Rendition(
        short_name='foo',
        verbose_name='Foo Rendition',
        description='Render TestModel instances as Foo.',
        path_to_template='anotherapp/foo.html',
        rendition_type='block'
    )
)
```

That's it! Remember: Rendition *short_name* values must be unique across all renditions associated with a Stuff class. If you try registering a rendition with the same *short_name* value as another registered rendition an *AlreadyRegisteredRendition* exception will raise.

2.2.3 Using the TextPlusStuff field

Using a TextPlusStuff field is easy just import it and set it to an attribute. Any options available to a django TextField (like blank=True) can be set on a TextPlusStuffField:

```
# someapp/models.py

from django.db import models

from textplusstuff.fields import TextPlusStuffField

class MyModel(models.Model):
    content = TextPlusStuffField()
```

TextPlusStuff fields store rich text as markdown and can serve it back as either raw markdown, plain text (formatting removed), or as HTML (markdown entities converted into HTML tags):

```
>>> from someapp.models import MyModel
>>> instance = MyModel(content='Oh _hello there_!')
>>> instance.save()
>>> instance.content.as_markdown()
```

(continues on next page)

(continued from previous page)

```
'Oh _hello there_!'
>>> instance.content.as_plaintext()
'Oh hello there!'
>>> instance.content.as_html()
'Oh <em>hello there</em>!'
```

Try pasting some tokens (that you find at /textplusstuff) into a TextPlusStuffField, saving the model instance associated with the field and then call the attributes above to see what happens.

2.2.3.1 Adding just-in-time extra context to .as_html() rendering

If you want to include extra context data beyond what is provided natively by a token just pass a dictionary to the *extra_context* keyword argument of the *as_html()* method:

```
>>> instance.content.as_html(extra_context={'some_key': 'some_value'})
```

This dictionary will then be passed to the context keyword argument of the serializer class associated with that token's Stuff config. Click here for more information about how to access this data within your serializer.

Automatically pass `extra_context` to all renditions associated with `Stuff`

If you'd like to automatically include the values passed to `extra_context` into your serializer context just use the `ExtraContextSerializerMixin` as one of your serializer superclasses.

Here's how we'd integrate it into the `TestModelSerializer` example:

```
# serializers.py

from rest_framework.serializers import ModelSerializer

from textplusstuff.serializers import ExtraContextSerializerMixIn

from .models import TestModel

class TestModelSerializer(ExtraContextSerializerMixIn,
                          ModelSerializer):

    class Meta:
        model = TestModel
        fields = (
            'name',
        )
```

Now any data passed like this: `instance.text_plus_stuff_field.as_html(extra_context={'foo': 'bar'})` will be available on all its renditions/templates at `{{ context.extra_content.foo }}` (where `{{ context.extra_content.foo }}` would be rendered as `bar`).

2.2.4 Admin Integration

There currently isn't a front-end interface for TextPlusStuff fields and this makes finding tokens unnecessarily difficult (unless you're a weirdo who likes groking JSON). To mitigate this, just swap the superclass of

your admin configurations from `django.contrib.admin.ModelAdmin` with `textplusstuff.admin.TextPlusStuffRegisteredModelAdmin` like so:

```
from django.contrib import admin

from textplusstuff.admin import TextPlusStuffRegisteredModelAdmin

# A model registered with textplusstuff.registry.stuff_registry
from .models import SomeModel

class SomeModelAdmin(TextPlusStuffRegisteredModelAdmin):
    # Configure like you would any admin.ModelAdmin class
    pass

admin.site.register(SomeModel, SomeModelAdmin)
```

This will add an 'Available Renditions' sections beneath the change/edit form within the admin that contains a table that lists all the available renditions for that model (including their instance-associated tokens).

2.3 Django REST Framework Integration

If you've got an API powered by Tom Christie's excellent Django REST Framework you can serve the content of a `TextPlusStuffField` simultaneously in a variety of formats with the `TextPlusStuffFieldSerializer`.

2.3.1 Example

To demonstrate how it works we'll use this simple model:

```
# myproject/content/models.py

from django.db import models

from textplusstuff.fields import TextPlusStuffField


class Content(models.Model):
    """Represents a piece of content."""
    content = TextPlusStuffField('Content')

    class Meta:
        verbose_name = 'Content Block'
        verbose_name_plural = 'Content Blocks'
```

OK, let's write a simple `ModelSerializer` subclass to serialize Content instances:

```
# myproject/content/serializers.py

from rest_framework import serializers

from textplusstuff.serializers import TextPlusStuffFieldSerializer

from .models import Content
```

(continues on next page)

(continued from previous page)

```
class ContentSerializer(serializers.ModelSerializer):
    """Serializes Content instances"""
    content = TextPlusStuffFieldSerializer()

    class Meta:
        model = Content
        fields = (
            'content',
        )
```

And here's what it would look like serialized:

```
>>> from myproject.content.models import Content
>>> content = Person.objects.create(
...     content="""# Oh hello!\n\nHere's some _italic_ and **bold** text."""
... )
>>> content.save()
>>> from myproject.content.serializers import ContentSerializer
>>> content_serialized = ContentSerializer(content)
>>> content_serialized.data
{
    "content": {
        "raw_text": "# Oh hello!\n\nHere's some _italic_ and **bold** text.", # The
↪'raw' content of the field as it is stored in the database.
        "as_plaintext": "Oh hello!\n\nHere's some italic and bold text.", # The␣
↪content of this field as plaintext (all markup/formatting and tokens removed)
        "as_markdown": "# Oh hello!\n\nHere's some _italic_ and **bold** text.", #␣
↪The content of this field as markdown (with tokens removed)
        "as_html": "<h1>Oh hello!</h1>\n\n<p>Here's some <em>italic</em> and <strong>
↪bold</strong> text.", # The content of this field as HTML with tokens rendered
        "as_html_no_tokens": "<h1>Oh hello!</h1>\n\n<p>Here's some <em>italic</em>␣
↪and <strong>bold</strong> text.", # The content of this field as HTML with tokens␣
↪removed
        "as_json": {
            "text_as_html": "<h1>Oh hello!</h1>\n\n<p>Here's some <em>italic</em> and
↪<strong>bold</strong> text.",
            "text_as_markdown": "# Oh hello!\n\nHere's some _italic_ and **bold**␣
↪text.",
            "content_nodes": []
        }
    }
}
```

Note: The example content used above doesn't include any tokens which is why the `'as_html'` and `'as_html_no_tokens'` as well as the `'raw_text'` and `'as_markdown'` values are identical.

2.4 Improving Performance

Docs coming soon!

2.4.1 Defining a `'constructed_field'`

2.4.2 Signal Architecture

CHAPTER 3

Release Notes

3.1 0.7

- Removed need to call findstuff() to discover stuff modules. Now we use Django's built in autodiscover_modules function and set everything up in the AppConfig.
- Removed Django 1.6.x compatibility

3.2 0.6

- Added Python 3.5 support
- Deprecated Python 3.3 support
- Added Django 1.9.x compatibility

3.3 0.5

- Added `as_json` method to `TextPlusStuffField`
- Added a new field (`TextPlusStuffConstructedField`) and signal (`update_constructed_fields`) that can be leveraged to improve speed/performance.

3.4 0.4.1

- Fixed a UnicodeDecodeError bug that arose in Python 2.7.5 when encoding text nodes that had non-ASCII encoded HTML entities.

3.5 0.4

- Added *ExtraContextSerializerMixin* for simplifying `extra_context`-to-serializer handoff.

3.6 0.3

- Added the ability to *register 'non-core' renditions* in a third-party application's already-registered Stuff class.
- `django-textplusstuff` is now available for installation via wheel.

3.7 0.2.1

- Squashed a bug that prevented `TextPlusStuffField` from serializing correctly (when using `dumpdata`).

3.8 0.2

- Added Django REST Framework *serialization* for `TextPlusStuffField`

3.9 0.1.3

- Fixed a Python 2.7.x-related encoding issue in the Stuff registry.

3.10 0.1.2

- Another `pip` installation hotfix: including template files in distribution.

3.11 0.1.1

- Squashed `pip` installation bug

3.12 0.1

- Initial open source release

CHAPTER 4

Roadmap to v1.0

- Support Django 1.10 - 2.0
- Create a javascript powered editor for writing markdown-flavored text and placing tokens.
- textplusstuff API POST support (so model instances registered with the stuff_registry can be created directly from a TextPlusStuff field widget)
- Create an example registered model to explain how the rendition/token architecture works.
- Document 'Constructed Field' functionality to improve performance.
- Document 'as_json' method on a TextPlusStuff instance.