django-markup Documentation

Release 1.0

Martin Mahner

May 01, 2023

Contents

This app is a generic way to provide filters that convert text into html. The idea was originally part of the django-template-utils package by James Bennet. I've encapsulated the markup system and expanded it into *django-markup*.

Contents:

CHAPTER 1

Installation

Download and install the package from the python package index (pypi):

```
pip install django-markup
```

Note that *django-markup* ships with some filters ready to use, the requirements of those filters are not installed by default. If you want to use all of the filters right away, you can install their latest packages with:

```
pip install django-markup[all_filter_dependencies]
```

1.1 Install the latest development version

The latest development version is available from GitHub:

```
git clone git://github.com/bartTC/django-markup.git
```

Install it with pipenv:

```
cd django-markup
pipenv install
```

Run the testsuite with pipenv as well:

```
pipenv run ./runtests.py
```

CHAPTER 2

Configuration

To install django-markup just add the package to your `INSTALLED_APPS` setting within your `settings.py`:

```
# settings.py
INSTALLED_APPS = (
    ...
    'django_markup',
)
```

Thats all. To start using it, see the following tutorials:

2.1 Usage in Django templates

django-markup provides a templatetag to apply filter on variables or text.

First make sure that in every template you want to use django-markup the template library *markup_tags* is loaded:

```
{% load markup_tags %}
```

Then apply the `apply_markup` on strings or variables you want to convert. The tag has one argument which defines the *Filter* you want to apply:

```
{{ entry.content|apply_markup:"markdown" }}
{{ "One line of a *string*"|apply_markup:"markdown" }}
```

Of course you can apply more than one filter:

```
{{ entry.content|apply_markup:"markdown"|apply_markup:"smartypants" }}
```

2.1.1 Multiline strings

You can use this filter for multiline strings too:

```
{% filter apply_markup:"markdown" %}
# Hello World #

I am a text that was converted with **markdown**!
{% endfilter %}
```

This results in:

```
<h1>Hello World</h1>
<p>I am a text that was converted with <strong>markdown</strong>!
```

2.1.2 A list of all bundled filters:

Creole Wiki Syntax

- Filter name: `creole`
- Pypi package: `python-creole`

This is a Python implementation of a parser for the Creole wiki markup language.

Linebreaks

- Filter name: `linebreaks`
- Provided by Django v1.0+

The `linebreaks` filter replaces line breaks in plain text with appropriate HTML; a single newline becomes an HTML line break (`<br />`) and a new line followed by a blank line becomes a paragraph break (`</p>`).

Markdown

- Filter name: `markdown`
- Pypi package: `Markdown`, `bleach`, `bleach-whitelist`

This filter comes with default settings:

```
{
    'safe_mode': True
}
```

You can override them by either subclassing the related `MarkupFilter` class or using the global settings:

```
MARKUP_SETTINGS = {
    'markdown': {
        'safe_mode': True
    }
}
```

None (no processing)

- Filter name: `none`

This is the most simple filter. It returns the text as is.

reStructuredText

- Filter name: `restructuredtext`
- Pypi package: `Docutils`

This filter converts a reStructuredText string to HTML. See a quick reference about reStructuredText.

This filter comes with default settings:

```
{
    'settings_overrides': {
        'raw_enabled': False,
        'file_insertion_enabled': False,
    }
}
```

You can override them by either subclassing the related `MarkupFilter` class or using the global settings:

```
MARKUP_SETTINGS = {
   'restructuredtext': {
      'settings_overrides': {
         'raw_enabled': True,
         'file_insertion_enabled': True,
      }
   }
}
```

Syntax Highlighting:

Pygments will automatically add a `code-block` directive with syntax highlighting.

rST Input:

```
Some **rST** text.

.. code-block:: python

    def test():
      return 'Hello World'
```

Output:

```
<div class="document">
<p>Some <strong>rST</strong> text.</p>
<pre class="code python literal-block">
<span class="keyword">def</span> <span class="name function">test</span><span class=
↪"punctuation">():</span>
  <span class="keyword">return</span> <span class="literal string single">\'Hello␣
↪World\'</span>
</pre>
</div>
```

By default, reStructuredText uses long class names. You can change the format of the class names using the syntax_highlight option:

```
MARKUP_SETTINGS = {
   'restructuredtext': {
      'settings_overrides': {
         'syntax_highlight': short,
      }
   }
}
```

Above output but with *short* option:

```
<div class="document">
<p>Some <strong>rST</strong> text.</p>
<pre class="code python literal-block">
<span class="keyword">def</span> <span class="name function">test</span><span class=
↪"punctuation">():</span>
  <span class="keyword">return</span> <span class="literal string single">\'Hello␣
↪World\'</span>
</pre>
</div>
```

SmartyPants

- Filter name: `smartypants`
- Pypi package: `SmartyPants`

SmartyPants is a free web publishing plug-in that easily translates plain ASCII punctuation characters into "smart" typographic punctuation HTML entities.

Textile

- Filter name: `textile`
- Pypi package: `Textile`

Textile is a lightweight markup language originally developed by Dean Allen and billed as a "humane Web text generator". Textile converts its marked-up text input to valid, well-formed XHTML and also inserts character entity references for apostrophes, opening and closing single and double quotation marks, ellipses and em dashes.

(Taken from Wikipedia)

Widont Filter

- Filter name: `widont`
- Provided by django-markup

Puts a non-breaking space ` ` within the last two words, to avoid orphan words in the next line.

2.2 Usage in Python code

For usage within python code, see the *MarkupFormatter* examples.

2.3 Usage in models

django-markup provides a `MarkupField`, a CharField that displays a list of *Filter*:

```
from django_markup.fields import MarkupField

class Entry(models.Model):
    content = models.TextField()
    markup = MarkupField(default='restructuredtext')
```

Usage in a template:

```
{% load markup_tags %}

{% for entry in entry_list %}
    {{ entry.content|apply_markup:entry.markup }}
{% endfor %}
```

The list of :ref:Filter can been overridden in your settings.py with a tuple called `MARKUP_CHOICES` which holds a list of filters. A default value would be:

```
MARKUP_CHOICES = (
    'none',
    'linebreaks',
    'markdown',
    'restructuredtext',
)
```

CHAPTER 3

Filter

A filter is a simple class that takes a text input, transforms it and returns the transformed text. New filters must abstract a base class `MarkupFilter`. Let's make an example that reads a text and converts it to uppercase:

```
from django_markup.filter import MarkupFilter

class UppercaseMarkupFilter(MarkupFilter):
    title = 'Uppercase text'
    def render(self, text, **kwargs):
        return text.upper()
```

A filter must contain a `render` method that takes a variable `text` as it's first argument which holds the text to transform. Also it must accept a `**kwargs` argument which is used for *Overriding filter settings*. At the end, the `render` method returns the modified text.

Additionally the filter class should contain an attribute `title` which holds a better human readable name for this filter.

Please have a look on the sourcecode of the bundled filters for better examples.

3.1 Adding filters to a formatter

See *MarkupFormatter* how to attach a filter to a formatter class.

3.2 A list of all bundled filters:

CHAPTER 4

MarkupFormatter

A `MarkupFormatter` instance is the central point that holds *Filter* and handles the text transformation. A generic instance of `MarkupFormatter` is already provided with django-markup and located in:

```
django_markup.markup.formatter
```

This instance takes 2 arguments, first the `text` to transform and a `filter_name` of the filter which should been used to convert the text. Example:

```
from django_markup.markup import formatter
print formatter('This is *markdown* text', filter_name='markdown')
```

You can pass any other keyword arguments which gets passed through the filter's render function. See this example:

```
from django_markup.markup import formatter
print formatter('This is *markdown* text', filter_name='markdown', safe_mode=True)
```

The `safe_mode=True` argument gets passed to the `render` method of the *Markdown* filter and at the end passed through the `markdown` function itself.

For a more generic use, see *Overriding filter settings*.

CHAPTER 5

Automatically loaded filter:

A bunch of *Filter* are loaded loaded automatically in the `django_markup.markup.formatter` class. Within your settings.py you can define which *Filter* are loaded from start. A default value would be:

```
from django_markup.filter.creole_filter import CreoleMarkupFilter
# other filter...

MARKUP_FILTER = {
    'creole': CreoleMarkupFilter,
    'linebreaks': LinebreaksMarkupFilter,
    'markdown': MarkdownMarkupFilter,
    'none': NoneMarkupFilter,
    'restructuredtext': RstMarkupFilter,
    'smartypants': SmartyPantsMarkupFilter,
    'textile': TextileMarkupFilter,
}
```

CHAPTER 6

Add a filter to a formatter instance:

To add a *Filter* to a `MarkupFormatter` instance, simply register it. Provide a filter name – the key that is used in the templatetag to define the filter – and the filter class to the HtmlFormatter `register` method:

```
from django_markup.markup import formatter
formatter.register('uppercase', UppercaseMarkupFilter)
```

6.1 Update a filter

If you want to update/replace a Filter class, use the `update` method of the `HtmlFormatter` instance. Similar to registering:

```
from django_markup.markup import formatter
formatter.update('uppercase', UppercaseMarkupFilter)
```

6.2 Remove a filter

To remove a filter from the `HtmlFormatter` instance, simply unregister it using the `unregister` method with it's filter name:

```
from django_markup.markup import formatter
formatter.unregister('uppercase')
```

CHAPTER 7

Bundled filters:

django-markup comes with a bunch of bundled filters to start right away. Currently they are:

CHAPTER 8

Advanced topic:

8.1 Overriding filter settings

8.1.1 Fallback filter

It's possible to set a *fallback* filter name. This filter is taken, if you provide no `filter_name` to the formatter-instance:

```
print formatter('This is *markdown* text', filter_name=None)

{{ entry.content|apply_markup:"" }}
```

In this case, add a variable in your `settings.py` called `MARKUP_FILTER_FALLBACK`:

```
MARKUP_FILTER_FALLBACK = 'linebreaks'
```

With this, the above examples would converted using the `linebreaks` filter.

8.1.2 Arguments to the markup filter

You can pass arguments to the markup-filter itself. Here is a real world example:

```
MARKUP_SETTINGS = {
    'restructuredtext': {
        'settings_overrides': {
            'initial_header_level': 2,
            'doctitle_xform': False,
            'footnote_references': 'superscript',
            'trim_footnote_reference_space': True,
            'default_reference_context': 'view',
            'link_base': ''
        }
```

(continues on next page)

(continued from previous page)

```
    }

    'markdown': {
        'safe_mode': True,
        'extensions': ('tables', )
    }
}
```

With the above setting, the call of the markdown function would like:

```
markdown.markdown(text, safe_mode=True, extensions=('tables',))
```

CHAPTER 9

Indices and tables

- genindex
- modindex
- search