tinymce Documentation

Release 3.0

Joost Cassee, Aljosa Mohorovic

March 20, 2017

Contents

django-tinymce is a Django application that contains a widget to render a form field as a TinyMCE editor.

CHAPTER 1

Quickstart

Make sure staticfiles application is properly configured (TinyMCE is bundled with django-tinymce and uses staticfiles to automatically serve TinyMCE files).

1. Install django-tinymce using pip (or any other way to install python package) from PyPI.

```
$ pip install django-tinymce
```

2. Add `tinymce` to INSTALLED_APPS in `settings.py` for your project:

```
INSTALLED_APPS = (
    ...
    'tinymce',
    ...
)
```

3. Add `tinymce.urls` to `urls.py` for your project:

```
urlpatterns = patterns('',
    ...
    (r'^tinymce/', include('tinymce.urls')),
    ...
)
```

4. Add a custom profile if needed in `settings.py`. A sample profile is shown below, for all the available options check http://www.tinymce.com/wiki.php/Configuration

```
TINYMCE_PROFILE = {
    'theme': 'modern',
    'plugins': 'noneditable advlist autolink link lists charmap hr searchreplace␣
↪wordcount visualblocks visualchars code insertdatetime save table contextmenu␣
↪directionality paste textcolor',
    'toolbar': 'undo redo | styleselect | bold italic | alignleft aligncenter␣
↪alignright alignjustify | bullist numlist outdent indent | forecolor backcolor␣
↪| upload_button',
    'noneditable_leave_contenteditable': 'true',
```

```
    'setup': 'addCustomButtons',
    'content_css': os.path.join(STATIC_URL, "mycss/tinymce.css"),
    'relative_urls': False,
    'remove_script_host': True,
    'document_base_url': APP_HOST_URL,
    'removed_menuitems': 'newdocument'
}
```

5. Use `HTMLField` where you would use `TextField` (or check Usage for alternatives).

```
from django.db import models
from tinymce.models import HTMLField
from myproject.settings import TINYMCE_PROFILE

class MyModel(models.Model):
    my_field = HTMLField(profile=TINYMCE_PROFILE)
```

The django-tinymce code is licensed under the MIT License. See the `LICENSE.txt` file in the distribution. Note that the TinyMCE editor is distributed under its own license. Note that django-tinymce and TinyMCE licenses are compatible (although different) and we have permission to bundle TinyMCE with django-tinymce.

CHAPTER 2

Documentation

Installation

This section describes how to install the django-tinymce application and related dependencies in your Django project.

Prerequisites

The django-tinymce application requires Django version 1.4 or higher.

If you want to use the spellchecker plugin using the supplied view you must install the PyEnchant package and dictionaries for your project languages. Note that the Enchant needs a dictionary that exactly matches your language codes. For example, a dictionary for code `'en-us'` will not automatically be used for `'en'`. You can check the availability of the Enchant dictionary for the `'en'` language code using the following Python code:

```
import enchant
enchant.dict_exists('en')
```

Note that the documentation will use 'TinyMCE' (capitalized) to refer the editor itself and 'django-tinymce' (lower case) to refer to the Django application.

Installation

1. Install django-tinymce using pip (or any other way to install python package) from PyPI. If you need to use a different way to install django-tinymce you can place the `tinymce` module on your Python path. You can put it into your Django project directory or run `python setup.py install` from a shell.

   ```
   $ pip install django-tinymce
   ```

2. Add `tinymce` to INSTALLED_APPS in `settings.py` for your project:

```
INSTALLED_APPS = (
    ...
    'tinymce',
    ...
)
```

3. Add `tinymce.urls` to `urls.py` for your project:

```
urlpatterns = patterns('',
    ...
    (r'^tinymce/', include('tinymce.urls')),
    ...
)
```

Configuration

The application can be configured by editing the project's `settings.py` file.

TINYMCE_JS_URL (default: settings.MEDIA_URL + 'js/tiny_mce/tiny_mce.js') The URL of the TinyMCE javascript file:

```
TINYMCE_JS_URL = os.path.join(MEDIA_ROOT, "path/to/tiny_mce/tiny_mce.js")
```

TINYMCE_JS_ROOT (default: settings.MEDIA_ROOT + 'js/tiny_mce') The filesystem location of the TinyMCE files. It is used by the compressor (see below):

```
TINYMCE_JS_ROOT = os.path.join(MEDIA_ROOT, "path/to/tiny_mce")
```

TINYMCE_DEFAULT_CONFIG (default: {'theme': "simple", 'relative_urls': False}) The default TinyMCE configuration to use. See the TinyMCE manual for all options. To set the configuration for a specific TinyMCE editor, see the `mce_attrs` parameter for the *widget*.

TINYMCE_SPELLCHECKER (default: False) Whether to use the spell checker through the supplied view. You must add `spellchecker` to the TinyMCE plugin list yourself, it is not added automatically.

TINYMCE_COMPRESSOR (default: False) Whether to use the TinyMCE compressor, which gzips all Javascript files into a single stream. This makes the overall download size 75% smaller and also reduces the number of requests. The overall initialization time for TinyMCE will be reduced dramatically if you use this option.

TINYMCE_FILEBROWSER (default: True if 'filebrowser' is in INSTALLED_APPS, else False) Whether to use the **django-filebrowser_** as a custom filebrowser for media inclusion. See the official TinyMCE documentation on custom filebrowsers.

Example:

```
TINYMCE_JS_URL = 'http://debug.example.org/tiny_mce/tiny_mce_src.js'
TINYMCE_DEFAULT_CONFIG = {
    'plugins': "table,spellchecker,paste,searchreplace",
    'theme': "advanced",
    'cleanup_on_startup': True,
    'custom_undo_redo_levels': 10,
}
TINYMCE_SPELLCHECKER = True
TINYMCE_COMPRESSOR = True
```

Usage

The application can enable TinyMCE for one form field using the `widget` keyword argument of `Field` constructors or for all textareas on a page using a view.

Using the widget

If you use the widget (recommended) you need to add some python code and possibly modify your template.

Python code

The TinyMCE widget can be enabled by setting it as the widget for a formfield. For example, to use a nice big TinyMCE widget for the content field of a flatpage form you could use the following code:

```
from django import forms
from django.contrib.flatpages.models import FlatPage
from tinymce.widgets import TinyMCE

class FlatPageForm(forms.ModelForm):
    ...
    content = forms.CharField(widget=TinyMCE(attrs={'cols': 80, 'rows': 30}))
    ...

    class Meta:
        model = FlatPage
```

The widget accepts the following extra keyword argument:

mce_attrs (default: {}) Extra TinyMCE configuration options. Options from `settings.TINYMCE_DEFAULT_CONFIG` (see *Configuration*) are applied first and can be overridden. Python types are automatically converted to Javascript types, using standard JSON encoding. For example, to disable word wrapping you would include `'nowrap': True`.

The tinymce application adds one TinyMCE configuration option that can be set using `mce_attrs` (it is not useful as a default configuration):

content_language (default: django.utils.translation.get_language_code()) The language of the widget content. Will be used to set the `language`, `directionality` and `spellchecker_languages` configuration options of the TinyMCE editor. It may be different from the interface language, which defaults to the current Django language and can be changed using the `language` configuration option in `mce_attrs`)

Templates

The widget requires a link to the TinyMCE javascript code. The `django.contrib.admin` templates do this for you automatically, so if you are just using tinymce in admin forms then you are done. In your own templates containing a TinyMCE widget you must add the following to the HTML `HEAD` section (assuming you named your form 'form'):

```
<head>
    ...
    {{ form.media }}
</head>
```

See also the section of form media in the Django documentation.

The `HTMLField` model field type

For lazy developers the tinymce application also contains a model field type for storing HTML. It uses the TinyMCE widget to render its form field. In this example, the admin will render the `my_field` field using the TinyMCE widget:

```
from django.db import models
from tinymce import models as tinymce_models

class MyModel(models.Model):
    my_field = tinymce_models.HTMLField()
```

In all other regards, `HTMLField` behaves just like the standard Django `TextField` field type.

Using the view

If you cannot or will not change the widget on a form you can also use the `tinymce-js` named view to convert some or all textfields on a page to TinyMCE editors. On the template of the page, add the following lines to the `HEAD` element:

```
{% load url from future %}
<script type="text/javascript" src="{{ MEDIA_URL }}js/tiny_mce/tiny_mce.js"></script>
<script type="text/javascript" src="{% url "tinymce-js" "NAME" %}"></script>
```

The `NAME` argument allows you to create multiple TinyMCE configurations. Now create a template containing the Javascript initialization code. It should be placed in the template path as `NAME/tinymce_textareas.js` or `tinymce/NAME_textareas.js`.

Example:

```
tinyMCE.init({
    mode: "textareas",
    theme: "advanced",
    plugins: "spellchecker,directionality,paste,searchreplace",
    language: "{{ language }}",
    directionality: "{{ directionality }}",
    spellchecker_languages : "{{ spellchecker_languages }}",
    spellchecker_rpc_url : "{{ spellchecker_rpc_url }}"
});
```

This example also shows the variables you can use in the template. The language variables are based on the current Django language. If the content language is different from the interface language use the `tinymce-js-lang` view which takes a language (`LANG_CODE`) argument:

```
{% load url from future %}
<script type="text/javascript" src="{% url "tinymce-js-lang" "NAME","LANG_CODE" %}"></
↪script>
```

External link and image lists

The TinyMCE link and image dialogs can be enhanced with a predefined list of links and images. These entries are filled using a variable loaded from an external Javascript location. The tinymce application can serve these lists for you.

Creating external link and image views

To use a predefined link list, add the `external_link_list_url` option to the `mce_attrs` keyword argument to the widget (or the template if you use the view). The value is a URL that points to a view that fills a list of 2-tuples (*name*, *URL*) and calls `tinymce.views.render_to_link_list`. For example:

Create the widget:

```
from django import forms
from django.core.urlresolvers import reverse
from tinymce.widgets import TinyMCE

class SomeForm(forms.Form):
    somefield = forms.CharField(widget=TinyMCE(mce_attrs={'external_link_list_url':␣
↪reverse('someapp.views.someview')}))
```

Create the view:

```
from tinymce.views import render_to_link_list

def someview(request):
    objects = ...
    link_list = [(unicode(obj), obj.get_absolute_url()) for obj in objects]
    return render_to_link_list(link_list)
```

Finally, include the view in your URLconf.

Image lists work exactly the same way, just use the TinyMCE `external_image_list_url` configuration option and call `tinymce.views.render_to_image_list` from your view.

The `flatpages_link_list` view

As an example, the tinymce application contains a predefined view that lists all `django.contrib.flatpages` objects: `tinymce.views.flatpages_link_list`. If you want to use a TinyMCE widget for the flatpages `content` field with a predefined list of other flatpages in the link dialog you could use something like this:

```
from django import forms
from django.core.urlresolvers import reverse
from django.contrib.flatpages.admin import FlatPageAdmin
from django.contrib.flatpages.models import FlatPage
from tinymce.widgets import TinyMCE

class TinyMCEFlatPageAdmin(FlatPageAdmin):
    def formfield_for_dbfield(self, db_field, **kwargs):
        if db_field.name == 'content':
            return db_field.formfield(widget=TinyMCE(
                attrs={'cols': 80, 'rows': 30},
                mce_attrs={'external_link_list_url': reverse('tinymce.views.flatpages_
↪link_list')},
            ))
        return super(TinyMCEFlatPageAdmin, self).formfield_for_dbfield(db_field,␣
↪**kwargs)

somesite.register(FlatPage, TinyMCEFlatPageAdmin)
```

If you want to enable this for the default admin site (`django.contrib.admin.site`) you will need to unregister the original ModelAdmin class for flatpages first:

```
from django.contrib import admin

admin.site.unregister(FlatPage)
admin.site.register(FlatPage, TinyMCEFlatPageAdmin)
```

The source contains a test project that includes this flatpages model admin. You just need to add the TinyMCE javascript code.

1. Checkout the test project: `svn checkout http://django-tinymce.googlecode.com/svn/trunk/testtinymce`
2. Copy the `tiny_mce` directory from the TinyMCE distribution into `media/js`
3. Run `python manage.py syncdb`
4. Run `python manage.py runserver`
5. Connect to http://localhost:8000/admin/

The TinyMCE preview button

TinyMCE contains a preview plugin that can be used to allow the user to view the contents of the editor in the website context. The tinymce application provides a view and a template tag to make supporting this plugin easier. To use it point the `plugin_preview_pageurl` configuration to the view named `tinymce-preview`:

```
from django.core.urlresolvers import reverse
widget = TinyMCE(mce_attrs={'plugin_preview_pageurl': reverse('tinymce-preview', "NAME
↪")})
```

The view named by `tinymce-preview` looks for a template named either `tinymce/NAME_preview.html` or `NAME/tinymce_preview.html`. The template accesses the content of the TinyMCE editor by using the `tinymce_preview` tag:

```
{% load tinymce_tags %}
<html>
<head>
...
{% tinymce_preview "preview-content" %}
</head>
<body>
...
<div id="preview-content"></div>
...
```

With this template code the tekst inside the HTML element with id `preview-content` will be replace by the content of the TinyMCE editor.

Custom buttons and plugins

You can use TinyMCE setup method to call custom JavaScript functions after all TinyMCE files are loaded but before the editor instance is rendered on the page. This is suitable to add custom buttons, menu entries or events to TinyMCE.

For example, to add custom button with simple functionality, add `setup` entry to your `TINYMCE_PROFILE`:

```
TINYMCE_PROFILE = {
    ...
    'setup': 'addCustomButtons',
```

```
    ...
}
```

Then load JavaScript file containing `addCustomButtons` function using Django ModelAdmin assets:

```
class ArticleAdmin(admin.ModelAdmin):
  class Media:
      js = ("tinymce_custom_buttons.js",)
```

`tinymce_custom_buttons.js` can look like this:

```
function addCustomButtons(editor) {
  // This function replaces all whitespaces in selected area with dashes
  var whitespace_replace = function () {
      var selected_content = editor.selection.getContent();
      var replaced_content = selected_content.replace(/ /g, "-");
      editor.selection.setContent(replaced_content);
      editor.undoManager.add();
  };
  // Adding button to editor
  editor.addButton('test-button', {
      tooltip: 'Replace whitespaces with dashes',
      icon: 'blockquote',
      onclick: whitespace_replace
  });
  // Adding menu item to editor
  editor.addMenuItem('test-button', {
      context: 'format',
      text: 'Replace whitespaces with dashes',
      icon: 'blockquote',
      onclick: whitespace_replace
  });
}
```

Note: for custom button to appear in editor toolbar you must explicitly append it to `TINYMCE_PROFILE` `toolbar` entry:

```
TINYMCE_PROFILE = {
    ...
    'setup': 'addCustomButtons',
    'toolbar': '... test-button',
    ...
}
```

For more information on custom buttons see official TinyMCE documentation.

If you want to implement more complex behavior (i.e. load auxiliary resources like button icons or js libraries) consider writing a TinyMCE plugin. TinyMCE plugins are small JavaScript files containing `tinymce.PluginManager.add` function (see TinyMCE plugin tutorial for more information). Then writing a plugin, place it in `tinymce/plugins/<your_plugin_name>/plugin.min.js` in one of your `STATICFILES_DIRS`, then load the plugin with `TINYMCE_PROFILE` (you still have to explicitly add button to a toolbar):

```
TINYMCE_PROFILE = {
    ...
    'plugins': '... example',
    'toolbar': '... example',
    ...
}
```

(don't forget to run `python manage.py collectstatic` if you are not using built-it Django development server). The button should appear on a toolbar.

Note: ensure you use unique plugin name. Overlapping of plugin names results in undefined behavior (see list of TinyMCE built-in plugins).

History

Changelog

Repository:

- Fixed mutable default argument in widget code.
- Fixed the external list example in the documentation.
- Fixed documentation and test code to use `db_field.formfield` instead of `forms.CharField`.

Release 1.5 (2009-02-13):

- Updated Google Code CSS location.
- Fixed a compressor crash when 'theme' configuration was omitted.
- Added a note in the documentation about Python-JSON type conversion.
- Fixed the filebrowser integration when serving media from a different domain.
- Fixed flatpages example code in documentation.
- Added support for the preview plugin.
- Added "'relative_urls': False" to the default settings to fix integration with django-filebrowser.

Release 1.4 (2009-01-28):

- Fixed bugs in compressor code causing it not to load.
- Fixed widget media property.

Release 1.3 (2009-01-15):

- Added integration with django-filebrowser.
- Added templates to source distribution.
- Updated TinyMCE compressor support: copying media files no longer required.

Release 1.2 (2008-11-26):

- Moved documentation from Wiki into repository.

Release 1.1 (2008-11-20):

- Added TinyMCE compressor support by Jason Davies.
- Added HTMLField.

Release 1.0 (2008-09-10):

- Added link and image list support.
- Moved from private repository to Google Code.

Credits

tinymce was written by Joost Cassee based on the work by John D'Agostino. It was partly taken from his code at the Django code wiki. The TinyMCE Javascript WYSIWYG editor is made by Moxiecode.

The TinyMCE compressor was written by Jason Davies based on the PHP TinyMCE compressor from Moxiecode.