Django Plugins

Developed by Alabian Solutions Ltd

How To Install Tinymce in Django 2.1

Introduction

TinyMCE is a WYSIWYG HTML editor, the word WYSIWYG is an acronym for "what you see is what you get". It is designed to simplify website content creation. One can easily copy HTML content and paste into TinyMCE editor, the content you paste will look exactly same, hence WYSIWYG. In this tutorial, we'll implement TinyMCE in Django admin and custom form.

Installation

There are many TinyMCE packages available for Django but we will use **django-tinymce4-lite**

Install it with pip:

```
pip install django-tinymce4-lite
```

Add **"tinymce"** to **INSTALLED_APPS** in **settings.py** of your project

```
INSTALLED_APPS = [
      ...
    'tinymce',
      ...
    ]
```

Add tinymce.urls to **urls.py** of your project

```
 urlpatterns = [
    ...
    path('tinymce/', include('tinymce.urls')),
    ...
```

```
]
```

Also in **settings.py** file add following code to configure TinyMCE's toolbars

```
TINYMCE_DEFAULT_CONFIG = {
    'height': 360,
    'width': 1120,
    'cleanup_on_startup': True,
    'custom_undo_redo_levels': 20,
    'selector': 'textarea',
    'theme': 'modern',
    'plugins': '''
            textcolor save link image media preview codesample contextmenu
            table code lists fullscreen  insertdatetime  nonbreaking
            contextmenu directionality searchreplace wordcount visualblocks
            visualchars code fullscreen autolink lists  charmap print  hr
            anchor pagebreak
            ''',
    'toolbar1': '''
            fullscreen preview bold italic underline | fontselect,
            fontsizeselect  | forecolor backcolor | alignleft alignright |
            aligncenter alignjustify | indent outdent | bullist numlist table
|
            | link image media | codesample |
            ''',
    'toolbar2': '''
            visualblocks visualchars |
```

```
            charmap hr pagebreak nonbreaking anchor |  code |
            ''',
    'contextmenu': 'formats | link image',
    'menubar': True,
    'statusbar': True,
    }
```

models.py

For brevity, we'll use minimal code required for this tutorial

```
from django.db import models
from tinymce import HTMLField


class Post(models.Model):
    title = models.CharField(max_length=120)
    description = models.TextField(max_length=250,null=True)
    content = HTMLField('Content')
    draft = models.BooleanField(default=False)


    def __str__(self):
        return self.title
```

Now we have to register Post model to admin panel, so to do that the **admin.py** should be like:

```
from django.contrib import admin

from . import models

admin.site.register(models.Post)
```

Apply database migration commands if you haven't, then open up admin panel and try adding a Post, TinyMCE should appear in the content field.

TinyMCE in modelform

If you want TinyMCE in your modelform then create a file **forms.py** in the same app and add following code

```
from django import forms

from tinymce import TinyMCE

from .models import Post

class TinyMCEWidget(TinyMCE):

    def use_required_attribute(self, *args):

        return False



class PostForm(forms.ModelForm):

    content = forms.CharField(

        widget=TinyMCEWidget(

            attrs={'required': False, 'cols': 30, 'rows': 10}

        )

    )


    class Meta:

        model = Post

        fields = '__all__'
```

views.py

```
from django.shortcuts import render

from revision_app.models import Post

from revision_app.forms import PostForm


def post_form(request):

      if request.method == 'POST':

            pst_form = PostForm(request.POST)


            if pst_form.is_valid():

                  pst_form.save()

      else:

            pst_form = PostForm()

            return render(request, 'front_end/post_form.html',
{'pt_form':pst_form})
```

urls.py in app

```
app_name = 'revision_app'

urlpatterns = [

    ……

    path('post_form_page/', views.post_form, name='post_form'),

   ……

]
```

And the template corresponding to modelform would be:

```
{% extends "base.html" %}

{% load static %}


{% block content %}


    {{ pt_form.media }}

    <div class="row form-error">

        <div class="column" id="content">

            <form  method="post" action='' enctype="multipart/form-data">

                {% csrf_token %}

                {{ pt_form.as_p  }}

                <input class="button" type="submit" value="Save">

            </form>

        </div>

    </div>


{% endblock %}
```

Post detail template

Now that we've configured TinyMCE, we need to render the content in post detail page, it can be done by applying safe filter to the content field of article in post detail template as illustrate below

```
{% extends "base.html" %}

{% load static %}

     {# assuming that your base.html contains block content    #}

     {#   and instance variable being supplied from view      #}

{% block content %}

            <h3 > {{instance.title}} </h3>

            <div> {{ instance.content | safe }}  </div>

{% endblock %}
```

Setting up filebrowser for media

In production, you need to define STATIC_ROOT and MEDIA_ROOT in settings.py for getting static files like CSS, JS etc. from third-party packages available to be served, so your project's settings.py should be configured like:

```
STATIC_ROOT = os.path.join(os.path.dirname(BASE_DIR),"static_cdn")

MEDIA_ROOT = os.path.join(os.path.dirname(BASE_DIR),"media")


STATIC_URL = '/static/'

MEDIA_URL = '/media/'

STATICFILES_DIRS= [

    STATIC_DIR

]
```

In **static_cdn**, all the static files from static directories of your project as well as third-party packages will be collected there, and in media_cdn you can use it you collect media objects like images, videos your project. For TinyMCE to store media files we need to create a directory named "**uploads**" in **media**. To upload an image with TinyMCE you can put a link to an image in upload image option, but to upload images or videos we need to install django filebrowser.

```
pip install django-filebrowser-no-grappelli
```

"django-filebrowser" package needs grappelli to work properly, grappelli is customized Django admin panel which doesn't look appealing at all, so we choose django-filebrowser-no-grappelli package which is compatible with Django's native admin panel as well as other bootstrapped admin panel as jet. For handling images TinyMCE requires pillow package, you can install it with pip

```
pip install pillow
```

Add filebrowser to INSTALLED_APPS in settings.py

```
INSTALLED_APPS = [

        ...

    'filebrowser',

        ...

]
```

Add the following code in project **urls.py**

```
from filebrowser.sites import site

urlpatterns = [

    ……

    path('admin/', admin.site.urls),

    path('admin/filebrowser/', site.urls),

    …..

]

if settings.DEBUG:

        urlpatterns += static(settings.STATIC_URL , document_root =
settings.STATIC_ROOT )
```

```
        urlpatterns += static(settings.STATIC_URL,
document_root=settings.STATIC_URL)

        urlpatterns += static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)
```

Finally, collect static files to static_cdn with the following command

```
python manage.py collectstatic
```

How to install Fontawesome

django-fontawesome is a Django app that provides a couple of Fontawesome/Django related utilities, namely:

- an IconField to associate Fontawesome icons with model instances
- templatetags to render Fontawesome icons

also included:

- admin support for the IconField
- fr locale translation

Settings

By default, django-fontawesome ships with and uses the lastest fontawesome release. You can configure django-fontawesome to use another release/source/cdn by specifying:

```
# default uses locally shipped version at 'fontawesome/css/font-
awesome.min.css'

FONTAWESOME_CSS_URL = '//cdn.example.com/fontawesome-min.css'  # absolute url

FONTAWESOME_CSS_URL = 'front_end/css/font-awesome.min.css'  # relative url
```

Installation / Usage

Install via pip:

```
pip install django-fontawesome
```

Add 'fontawesome' to your installed apps setting like this:

```
INSTALLED_APPS = (

    ...

    'fontawesome',

)
```

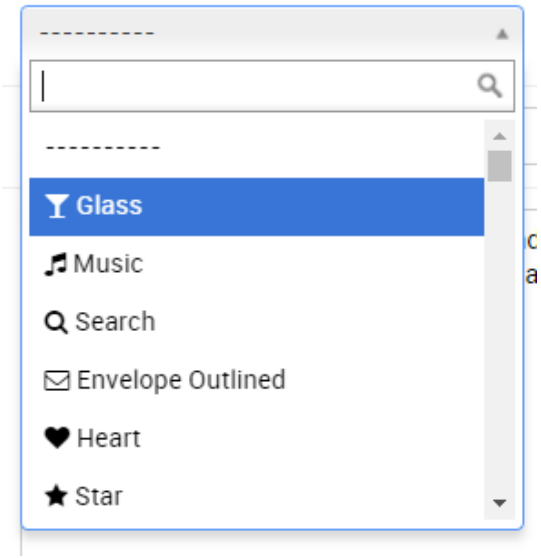
Import and use the IconField:

```
from fontawesome.fields import IconField


class Category(models.Model):

    ...

    icon = IconField()
```



You can then render the icon in your template like this:

```
{% for category in categories.all %}

    {% if category.icon %}

        {{ category.icon.as_html }}

    {% endif %}

{% endfor %}
```

django-fontawesome ships with two template

- tags, fontawesome_stylesheet and fontawesome_icon.
- the former inserts a stylesheet link with a pre-configured href according to the FONTAWESOME_CSS_URL setting

- the latter renders icons, and accepts the following optional keywords arguments: large, spin, fixed, li, border: (true/false), rotate: (90/180/270), title: (string)
- you can also colorize an icon using the color='red' keyword argument to the fontawesome_icon template tag
- example usage:

```
{% load fontawesome %}

<head>

  {% fontawesome_stylesheet %}

  ...

</head>

{% fontawesome_icon 'user' color='red' %}

{% fontawesome_icon 'star' large=True spin=True %}

<ul class="fa-ul">

   <li> {% fontawesome_icon 'home' rotate=90 li=True %} One</li>

</ul>
```