django-jinja Documentation

Release 0.23.1

Andrey Antukh

February 25, 2014

Contents

Release v0.23.1.

django-jinja is a *BSD Licensed*, simple and nonobstructive jinja2 integration with Django.

CHAPTER 1

Introduction

Jinja2 provides certain advantages over the native system of Django, for example, explicit calls to callable from templates, has better performance and has a plugin system, etc ...

There are other projects that attempt do same thing: Djinja, Coffin, etc... Why one more?

- Unlike Djinja, **django-jinja** is not intended to replace a Django template engine, but rather, it complements the Django's template engine, giving the possibility to use both.
- Unlike Coffin, the django-jinja codebase is much smaller and more modern. This way is much more maintainable and easily understandable how the library works.

CHAPTER 2

Features

- Auto-load templatetags compatible with Jinja2 on same way as Django.
- Django templates can coexist with Jinja2 templates without any problems. It works as middleware, intercepts Jinja templates by file path pattern.
- Django template filters and tags can mostly be used in Jinja2 templates.
- I18n subsystem adapted for Jinja2 (makemessages now collects messages from Jinja templates)
- Compatible with python2 and python3 using same codebase.
- Supported django versions: 1.4, 1.5, 1.6+

New in version 0.13: Regex template intercept (it gives a lot of flexibility with slighty performance decrease over a default intercept method).

New in version 0.21: Optional support for bytecode caching that uses Django's built-in cache framework by default.

CHAPTER 3

User guide

3.1 Quickstart

3.1.1 Install

You can download tarball from Pypi, extract this and install with:

```
tar xvf django-jinja-x.y.tar.gz
cd django-jinja
python setup.py install
```

Other (recomended) alternative is install with **pip**:

```
pip install django-jinja
```

3.1.2 Configure

The first step is replace django **TEMPLATE_LOADERS** with a django-jinja adapted loaders, and put `django_jinja` on **INSTALLED_APPS** tuple.

```
TEMPLATE_LOADERS = (
    'django_jinja.loaders.AppLoader',
    'django_jinja.loaders.FileSystemLoader',
)

INSTALLED_APPS += ('django_jinja',)
```

django-jinja template loaders inherit's from a django template loaders and add some condition for render jinja templates.

This condition is very simple. Basically it depends on the file extension, files with `.html` extension are rendered with django template engine and files with `.jinja` extension are rendered with jinja2 template engine.

You can specify the default extension for jinja2 with this settings:

```
DEFAULT_JINJA2_TEMPLATE_EXTENSION = '.jinja'
```

With **0.13** version, **django-jinja** incorporates more flexible method for intercept templates, using regex for matching.

Note: this method has worse perfomance that the default intercept method (by extension):

```
# Same behavior of default intercept method
# by extension but using regex (not recommended)
DEFAULT_JINJA2_TEMPLATE_INTERCEPT_RE = r'.*jinja$'

# More advanced method. Intercept all templates
# except from django admin.
DEFAULT_JINJA2_TEMPLATE_INTERCEPT_RE = r"^(?!admin/).*"
```

3.1.3 Optional settings

Additionally, **django-jinja** exposes some other settings parameters for costumize your jinja2 environment:

JINJA2_ENVIRONMENT_OPTIONS

Low level kwargs parameters for a jinja2 `Environment` instance. Example usage:

```
JINJA2_ENVIRONMENT_OPTIONS = {
    'block_start_string' : '\BLOCK{',
    'block_end_string' : '}',
    'variable_start_string' : '\VAR{',
    'variable_end_string' : '}',
    'comment_start_string' : '\#{',
    'comment_end_string' : '}',
    'line_statement_prefix' : '%-',
    'line_comment_prefix' : '%#',
    'trim_blocks' : True,
    'autoescape' : False,
}
```

JINJA2_AUTOESCAPE

Boolean value that enables or disables template autoescape. Default value is `True`

JINJA2_MUTE_URLRESOLVE_EXCEPTIONS

Boolean value that mute reverse url exceptions produced by url tag. Defaul value is `False`

JINJA2_FILTERS_REPLACE_FROM_DJANGO

Boolean value that enables overwrite some jinja filters with django filters. Defaulg value is `True`

3.1.4 Bytecode caching

django-jinja supports the use of Jinja2's template bytecode caching system to improve performance. It includes a default Jinja2 bytecode cache implementation that makes use of Django's built-in cache framework. This way, the template bytecode can be stored into a Django cache backend configured in your project via the CACHES setting. However, you can also use your own Jinja2 bytecode cache class.

JINJA2_BYTECODE_CACHE_ENABLE

A boolean value that enables or disables bytecode caching. Defaults to `False`.

JINJA2_BYTECODE_CACHE_NAME

The name of the Django cache backend to use for storing the template bytecode, as defined in your CACHES setting. Defaults to `'default'`.

JINJA2_BYTECODE_CACHE_BACKEND

A dotted path to a Jinja2 bytecode cache class. See the Jinja2 docs for reference if you want to implement your own cache class. Defaults to the backend built in to **django-jinja** (`'django_jinja.cache.BytecodeCache'`).

3.2 Differences

In django, creating new tags is simpler than in Jinja2. You should remember that in jinja tags are really extensions and have a different purpose than the django template tags.

Thus for many things that the django template system uses tags, django-jinja will provide functions with the same functionality.

3.2.1 Reverse urls on templates

In django you are accustomed to using the url tag:

```
{% url 'ns:name' pk=obj.pk %}
```

With jinja, you can use **reverseurl** filter or **url** global function. For example:

```
{{ 'ns:name'|reverseurl(pk=obj.pk) }}
{{ url('ns:name', pk=obj.pk) }}
```

3.2.2 Static files tag

On modern django apps we are accustomed to seeing a **static** template tag:

```
{% load static from staticfiles %}
{% static "js/lib/foo.js" %}
{% static "js/lib/foo.js" as staticurl %}
```

Jinja exposes a **static** global function or a **static** filter which does the same thing:

```
{{ "js/lib/foo.js"|static }}
{{ static("js/lib/foo.js" ) }}
{% set staticurl = static("js/lib/foo.js" ) %}
```

3.2.3 I18N and Django gettext

django-jinja has a built-in extension to the `makemessages` command, that correctly collects messages from jinja templates.

Here is an example:

```
python manage.py makemessages -a -e py,jinja,html
```

```
{{ _('i18n data') }}
{% trans %}
    i18n data
{% endtrans %}
```

3.2.4 Register global functions

You can register your global functions as you are registering template tags or filters in django.

Simple example:

```
# <someapp>/templatetags/<anyfile>.py
from django_jinja import library

lib = library.Library()

@lib.global_function
def myupper(name):
    return name.upper()
```

Functions, filters, or tests are registered globally on jinja automatically, without an explicit load templatetag.

3.2.5 Render 4xx/500 pages with jinja

Because django-jinja works as middleware that intercepts template rendering, standard django sepecial handlers (views) do not use jinja to render 404, 403 or 500 pages. To fix this, you can define your own views or use django-jinja's predefined ones.

Example:

```
# Your main urls.py
from django_jinja import views

handler403 = views.PermissionDenied.as_view()
handler404 = views.PageNotFound.as_view()
handler500 = views.ServerError.as_view()
```

3.3 Contrib modules

django-jinja comes with integration with other applications in django.

At the moment, it only has one contrib app, but in future it can integrate with others.

3.3.1 django-pipeline

Pipeline is an asset packaging library for Django (official description).

To activate this plugin add `django_jinja.contrib._pipeline` to your `INSTALLED_APPS` tuple:

```
INSTALLED_APPS += ('django_jinja.contrib._pipeline',)
```

Now, you can use `compressed_css` and `compressed_js` as global functions:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Foo</title>
        {{ compressed_css("main") }}
    </head>
    <body>
        <!-- body -->
```

```
    </body>
</html>
```

3.3.2 easy_thumbnails

Easy Thumbnails is a thumbnail generation library for Django.

To activate this plugin add `django_jinja.contrib._easy_thumbnails` to your `INSTALLED_APPS` tuple:

```
INSTALLED_APPS += ('django_jinja.contrib._easy_thumbnails',)
```

Now, you can use the `thumbnail` global function:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Foo</title>
    </head>
    <body>
        <img src="{{ thumbnail(file, size=(400, 400)) }}">
    </body>
</html>
```

3.3.3 django-subdomains

Subdomain helpers for the Django framework, including subdomain-based URL routing.

To activate this plugin add `django_jinja.contrib._subdomains` to your `INSTALLED_APPS` tuple:

```
INSTALLED_APPS += ('django_jinja.contrib._subdomains',)
```

Now you can use the `url` global function with the subdomain parameter:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Foo</title>
    </head>
    <body>
        <img src="{{ url('homepage', subdomain='wildcard') }}">
    </body>
</html>
```

3.3.4 dajax-ice

Easy to use AJAX library for django.

First, follow the install instructions in Dajaxice Quickstart.

Then, activate this plugin by adding `django_jinja.contrib._dajaxice` to your `INSTALLED_APPS` tuple:

```
INSTALLED_APPS = (
#...,
#...,
#...,
'django_jinja',
```

```
'django_jinja.contrib._pipeline',
'django_jinja.contrib._dajaxice',
'dajaxice',
)
```

Now you can use the `dajaxice_js_import` global context function:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Foo</title>
        {{ dajaxice_js_import() }}
    </head>
    <body>
        ...
    </body>
</html>
```

Troubleshooting

- ImportError: No module named defaults
  - try downgrading to Django 1.5.x