Django Meta Documentation

Release 2.4.3.dev1

Iacopo Spalletti

Apr 16, 2024

CONTENTS

A pluggable app allows Django developers to quickly add meta tags and OpenGraph, Twitter, and Schema.org properties to their HTML responses.

> **Warning:** **INCOMPATIBLE CHANGE**: as of version 2.0 django-meta has no longer supports Google+, basic Schema.org support has been introduced.

CHAPTER

ONE

USAGE

django meta has two different operating mode:

- *Model support*
- *View support*

1.1 Installation

- Install using pip:

```
pip install django-meta
```

- Add `meta` to `INSTALLED_APPS`:

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

- Add the *required namespace tags* to your base template for compliance with metadata protocols.
- Optionally you can install and configure sekizai

1.2 Upgrading

When upgrading from version 1.x to 2.0 you must

- Replace `META_GPLUS_TYPE` with `META_SCHEMAORG_TYPE`;
- Replace `META_USE_GPLUS_PROPERTIES` with `META_USE_SCHEMAORG_PROPERTIES`;
- Remove all references to `gplus_author`, `gplus_publisher`;
- Replace all `gplus_title`, `gplus_description`, `gplus_type`, `use_gplus` with the corresponding `schemaorg` attributes`;
- Replace all `googleplus_prop`, `googleplus_html_scope`, `googleplus_scope` with the corresponding `schemaorg` templatetags;

1.3 Model support

1.3.1 Concepts

django-meta provides a mixin to handle metadata in your models.

Actual data are evaluated at runtime pulling values from model attributes and methods.

To use it, defines a `_metadata` attribute as a dictionary of tag/value pairs;

- **tag** is the name of the metatag as used by `meta.html` template
- **value** is a string that is evaluated in the following order:
  - model method name called with the meta attribute as argument
  - model method name called with no arguments
  - model attribute name (evaluated at runtime)
  - string literal (if none of the above exists)

If **value** is `False` or it is evaluated as `False` at runtime the tag is skipped.

To use this mixin you must invoke `as_meta()` on the model instance for example in the get_context_data().

You can also add custom tags / properties. See *Adding custom tags / properties* for details.

Request

`as_meta()` accepts the `request` object that is saved locally and is available to methods by using the `get_request` method.

Public interface

`ModelMeta.get_meta(request=None)`: returns the metadata attributes definition. Tipically these are set in `_metadata` attribute in the model;

`ModelMeta.as_meta(request=None)`: returns the meta representation of the object suitable for use in the template;

`ModelMeta.get_request()`: returns the `request` object, if given as argument to `as_meta`;

`ModelMeta.get_author()`: returns the author object for the current instance. Default implementation does not return a valid object, this **must** be overridden in the application according to what is an author in the application domain;

`ModelMeta.build_absolute_uri(url)`: create an absolute URL (i.e.: complete with protocol and domain); this is generated from the `request` object, if given as argument to `as_meta`;

1.3.2 Usage

1. Configure `django-meta` according to documentation
2. Add meta information to your model:

```
from django.db import models
from meta.models import ModelMeta

class MyModel(ModelMeta, models.Model):
```

(continues on next page)

(continued from previous page)

```
    name = models.CharField(max_length=20)
    abstract = models.TextField()
    image = models.ImageField()
    ...

    _metadata = {
        'title': 'name',
        'description': 'abstract',
        'image': 'get_meta_image',
        ...
    }
    def get_meta_image(self):
        if self.image:
            return self.image.url
```

3. Push metadata in the context using `as_meta` method:

```
class MyView(DetailView):

    ...

    def get_context_data(self, **kwargs):
        context = super(MyView, self).get_context_data(self, **kwargs)
        context['meta'] = self.get_object().as_meta(self.request)
        return context
```

4. For Function Based View can just use `as_meta()` method for pass "meta" context variable:

```
def post(request, id):
    template = 'single_post.html'
    post = Post.objects.get(pk=id)
    context = {}
    context['post'] = post
    context['meta'] = post.as_meta()
    return render(request, template, context)
```

5. Include `meta/meta.html` template in your templates:

```
{% load meta %}

<html>
<head {% meta_namespaces %}>
    {% include "meta/meta.html" %}
</head>
<body>
</body>
</html>
```

Note

- For OpenGraph / Facebook support, edit your <head> tag to use `meta_namespaces` templatetags

1.3.3 Reference template

See below the basic reference template:

```
{% load meta %}

<html>
<head {% meta_namespaces %}>
    {% include "meta/meta.html" %}
</head>
<body>
{% block content %}
{% endblock content %}
</body>
</html>
```

1.4 View support

1.4.1 Using the view

You render the meta tags by including a `meta.html` partial template in your view templates. This template will only render meta tags if it can find a `meta` object in the context, so you can safely include it in your base template to have it render on all your pages.

The `meta.html` template expects to find an object called `meta` in the template context which contains any of the following attributes:

- use_og
- use_twitter
- use_facebook
- use_schemaorg
- use_title_tag
- title
- og_title
- twitter_title
- schemaorg_title
- description
- keywords
- url
- image
- image_object

- image_width
- image_height
- object_type
- site_name
- twitter_site
- twitter_creator
- twitter_type
- facebook_app_id
- locale
- extra_props
- extra_custom_props

In all cases, if the attribute is not set/empty, the matching metadata/property is not rendered.

Note: attribute `twitter_card` is available as deprecated attribute with the same meaning of `twitter_type`. It will be removed in version 3.0, so update your code accordingly.

Meta object

The core of django-meta is the `Meta` class. Although you can prepare the metadata for the template yourself, this class can make things somewhat easier.

To set up a meta object for use in templates, simply instantiate it with the properties you want to use:

```
from meta.views import Meta

meta = Meta(
    title="Sam's awesome ponies",
    description='Awesome page about ponies',
    keywords=['pony', 'ponies', 'awesome'],
    extra_props = {
        'viewport': 'width=device-width, initial-scale=1.0, minimum-scale=1.0'
    },
    extra_custom_props=[
        ('http-equiv', 'Content-Type', 'text/html; charset=UTF-8'),
    ]
)
```

When the time comes to render the template, you **must** include the instance as `'meta'` context variable. In case you use class-based views check the *view mixin* helper, for function based views you must pass the `meta` object manually to the context where needed.

`Meta` also accept an (optional) `request` argument to pass the current request, which is used to retrieve the `SITE_ID` if it's not in the settings.

The `Meta` instances have the same properties as the keys listed in the *Using the view* section. For convenience, some of the properties are 'smart', and will modify values you set. These properties are:

- keywords

- url
- image
- image_object

For brevity, we will only discuss those here.

Meta.keywords

When you assign keywords either via the constructor, or by assigning an iterable to the `keywords` property, it will be cleaned up of all duplicates and returned as a `set`. If you have specified the *META_INCLUDE_KEYWORDS*, the resulting set will also include them. If you omit this argument when instantiating the object, or if you assign `None` to the `keywords` property, keywords defined by *META_DEFAULT_KEYWORDS* setting will be used instead.

Meta.url

Setting the url behaves differently depending on whether you are passing a path or a full URL. If your URL starts with `'http'`, it will be used verbatim (not that the actual validity of the url is not checked so `'httpfoo'` will be considered a valid URL). If you use an absolute or relative path, domain and protocol parts would be prepended to the URL. Here's an example:

```
m = Meta(url='/foo/bar')
m.url  # returns 'http://example.com/foo/bar'
```

The actual protocol and domain are dependent on the *META_SITE_PROTOCOL* and *META_SITE_DOMAIN* settings. If you wish to use the Django's sites contrib app to calculate the domain, you can either set the *META_USE_SITES* setting to `True`, or pass the `use_sites` argument to the constructor:

```
m = Meta(url='/foo/bar', use_sites=True)
```

Note that using the sites app will trigger database queries and/or cache hits, and it is therefore disabled by default.

Meta.image_object

The `image_object` property is the most complete way to provide image meta.

To use this property, you must pass a dictionary with at least the `url` attribute.

All others keys will be rendered alongside the `url`, if the specific protocol provides it.

Currently only OpenGraph support more than the image url, and you might add:

- `width`: image width
- `height`: image height
- `alt`: alternate image description
- `secure_url`: https URL for the image, if different than the `url` key
- `type`: image mime type

example:

```
media = {
    'url': 'http://meta.example.com/image.gif',
    'secure_url': 'https://meta.example.com/custom.gif',
    'type': 'some/mime',
    'width': 100,
    'height': 100,
    'alt': 'a media',
}
```

it will be rendered as:

```
<meta property="og:image:alt" content="a media">
<meta property="og:image:height" content="100">
<meta property="og:image:secure_url" content="https://meta.example.com/image.gif">
<meta property="og:image:type" content="some/mime">
<meta property="og:image:url" content="http://meta.example.com/image.gif">
<meta property="og:image:width" content="100">
```

Meta.image

The `image` property behaves the same way as `url` property with one notable difference. This property treats absolute and relative paths differently. It will place relative paths under the *META_IMAGE_URL*.

if `image_object` is provided, it takes precedence over this property, for all the protocols, even if they only support the image URL.

View mixin

As a convenience to those who embrace the Django's class-based views, django-meta includes a mixin that can be used with your views. Using the mixin is very simple:

```
from django.views.generic import View

from meta.views import MetadataMixin


class MyView(MetadataMixin, View):
    title = 'Some page'
    description = 'This is an awesome page'
    image = 'img/some_page_thumb.gif'
    url = 'some/page/'

    ....
```

The mixin sports all properties listed in the *Using the view* section with a few additional bells and whistles that make working with them easier. The mixin will return an instance of the `Meta` class (see *Meta object*) as `meta` context variable. This is, in turn, used in the partial template to render the meta tags (see *Rendering meta tags*).

Each of the properties on the mixin can be calculated dynamically by using the `MetadataMixin.get_meta_PROPERTYNAME` methods, where `PROPERTYNAME` is the name of the property you wish the calculate at runtime. Each method will receive a `context` keyword argument containing the request context.

For example, to calculate the description dynamically, you may use the mixin like so:

```
class MyView(MetadataMixin, SingleObjectMixin, View):
    ...

    def get_meta_description(self, context):
        return self.get_object().description
```

There are two more methods that you can overload in your view classes, and those are `get_domain` and `get_protocol`.

Reference template

See below the basic reference template:

```
{% load sekizai_tags meta %}

<html {% render_block 'html_extra' %}>
<head {% meta_namespaces %}>
    {{ meta.og_description }}
    {% include "meta/meta.html" %}
</head>
<body>
{% block content %}
{% endblock content %}
</body>
</html>
```

Properties

use_og

This key contains a boolean value, and instructs the template to render the OpenGraph properties. These are usually used by FaceBook to get more information about your site's pages.

use_twitter

This key contains a boolean value, and instructs the template to render the Twitter properties. These are usually used by Twitter to get more information about your site's pages.

use_facebook

This key contains a boolean value, and instructs the template to render the Facebook properties. These are usually used by Facebook to get more information about your site's pages.

use_schemaorg

This key contains a boolean value, and instructs the template to render the Google+. These are usually used by Google to get more information about your site's pages.

use_title_tag

This key contains a boolean value, and instructs the template to render the `<title></title>` tag. In the simple case, you use `<title></title>` tag in the templates where you can override it, but if you want to generate it dynamically in the views, you can set this property to `True`.

title

This key is used in the `og:title` OpenGraph property if `use_og` is `True`, `twitter:title` if `use_twitter` is `True`, `itemprop="title"` if `use_schemaorg` is `True` or `<title></title>` tag if `use_title_tag` is `True`.

The service-specific variants are also supported:

- `og_title`
- `twitter_title`
- `schema_title`

If set on the `Meta` object, they will be used instead of the generic title which will be used as a fallback.

description

This key is used to render the `description` meta tag as well as the `og:description` and `twitter:description` property.

keywords

This key should be an iterable containing the keywords for the page. It is used to render the `keywords` meta tag.

url

This key should be the *full* URL of the page. It is used to render the `og:url`, `twitter:url`, `itemprop=url` property.

image_object

This key must be set to a dictionary containing at least the `url` key, additional keys will be rendered if supported by each protocol. Currently only OpenGraph supports additional image properties.

Example:

```
media = {
    'url': 'http://meta.example.com/image.gif',
    'secure_url': 'https://meta.example.com/custom.gif',
    'type': 'some/mime',
```

(continues on next page)

(continued from previous page)

```
    'width': 100,
    'height': 100,
    'alt': 'a media',
}
```

image

This key should be the *full* URL of an image to be used with the `og:image`, `twitter:image`, `itemprop=image` property.

image_width

This key should be the width of image. It is used to render `og:image:width` value

image_height

This key should be the height of image. It is used to render `og:image:height` value

object_type

This key is used to render the `og:type` property.

site_name

This key is used to render the `og:site_name` property.

twitter_site

This key is used to render the `twitter:site` property.

twitter_creator

This key is used to render the `twitter:creator` property.

twitter_type

This key is used to render the `twitter:card` property.

facebook_app_id

This key is used to render the `fb:app_id` property.

locale

This key is used to render the `og:locale` property.

extra_props

A dictionary of extra optional properties:

```
{
    'foo': 'bar',
    'key': 'value'
}

...

<meta name="foo" content="bar">
<meta name="key" content="value">
```

See *Adding custom tags / properties* for details.

extra_custom_props

A list of tuples for rendering custom extra properties:

```
[
    ('key', 'foo', 'bar')
    ('property', 'name', 'value')
]

...

<meta name="foo" content="bar">
<meta property="name" content="value">
```

1.5 schema.org

`django-meta` provides full support for schema.org in JSON-LD format.

schema.org is supported in both *Model support* and *View support* framework.

1.5.1 Model-level

In the same way as basic *_metadata* attribute, *_schema* exists to resolve and build the per-object **Schema.org** representation of the current object.

As per *_metadata*, *_schema* values can contains the name of a method, property or attribute available on the class:

```
class Blog(ModelMeta, Model)
    ...
    _schema = {
        'image': 'get_image_full_url',
        'articleBody': 'text',
        'articleSection': 'get_categories',
        'author': 'get_schema_author',
        'copyrightYear': 'copyright_year',
        'dateCreated': 'get_date',
        'dateModified': 'get_date',
        'datePublished': 'date_published',
        'headline': 'headline',
        'keywords': 'get_keywords',
        'description': 'get_description',
        'name': 'title',
        'url': 'get_full_url',
        'mainEntityOfPage': 'get_full_url',
        'publisher': 'get_site',
    }
```

1.5.2 View-level

Meta and *MetadataMixin* provides a few API to work with **schema.org** properties.

MetadataMixin

The high level interface is *meta.views.MetadataMixin.get_schema()* which works in much the same way as *meta.models.ModelMeta._schema*.

In *get_schema()* you must return the whole **schema.org** structure.

For a single object it can look like this:

```
def get_schema(self, context=None):
    return {
        'image': self.object.get_image_full_url(),
        'articleBody': self.object.text,
        'articleSection': self.object.get_categories(),
        'author': self.object.get_schema_author(),
        'copyrightYear': self.object.date_published.year,
        'dateCreated': self.object.get_date(),
        'dateModified': self.object.get_date(),
        'datePublished': self.object.date_published(),
        'headline': self.object.abstract[:50],
        'keywords': self.object.get_keywords(),
        'description': self.object.get_description(),
```

(continues on next page)

(continued from previous page)

```
        'name': self.object.title(),
        'url': self.object.get_full_url(),
        'mainEntityOfPage': self.object.get_full_url(),
        'publisher': self.object.get_site(),
    }
```

Note: as it's *schema* responsibility to convert objects to types suitable for json encoding, you are not required to put only literal values here. Instances of *Meta*, dates, iterables and dictionaries are allowed.

Meta

The low level interface is *meta.views.Meta._schema()* attribute or (schema argument to *Meta* constructor):

```
class MyView(View):

    def get_context_data(self, **kwargs):
        context = super(PostDetailView, self).get_context_data(**kwargs)
        context['meta'] = Meta(schema={
            '@type': 'Organization',
            'name': 'My Publisher',
            'logo': Meta(schema={
                '@type': 'ImageObject',
                'url': self.get_image_full_url()
            })
        })
        return context
```

1.6 Settings

django-meta has a few configuration options that allow you to customize it. Two of them are required: `META_SITE_PROTOCOL` and `META_SITE_DOMAIN`. By default, if they are unset, an `ImproperlyConfigured` exception will be raised when dealing with `url` and `image` properties. You can either set them, or overload the `Meta` class' `get_domain` and `get_protocol` methods (see *Meta object* section).

Warning: **INCOMPATIBLE CHANGE**: as of version 2.0 django-meta has no longer supports Google+, basic Schema.org support has been introduced.

1.6.1 META_SITE_PROTOCOL

Defines the protocol used on your site. This should be set to either `'http'` or `'https'`. Default is `None`.

1.6.2 META_SITE_DOMAIN

Domain of your site. The `Meta` objects can also be made to use the Django's Sites framework as well (see *Meta object* and *META_USE_SITES* sections). Default is `None`.

1.6.3 META_SITE_TYPE

The default `og:type` property to use site-wide. You do not need to set this if you do not intend to use the OpenGraph properties. Default is `None`.

1.6.4 META_SITE_NAME

The site name to use in `og:site_name` property. Althoug this can be set per view, we recommend you set it globally. Defalt is `None`.

1.6.5 META_INCLUDE_KEYWORDS

Iterable of extra keywords to include in every view. These keywords are appended to whatever keywords you specify for the view, but are not used at all if no keywords are specified for the view. See *META_DEFAULT_KEYWORDS* if you wish to specify keywords to be used when no keywords are supplied. Default is `[]`.

1.6.6 META_DEFAULT_KEYWORDS

Iterable of default keywords to use when no keywords are specified for the view. These keywords are not included if you specify keywords for the view. If you need keywords that will always be present, regardless of whether you've specified any other keywords for the view or not, you need to combine this setting with *META_INCLUDE_KEYWORDS* setting. Default is `[]`.

1.6.7 META_IMAGE_URL

This setting is used as the base URL for all image assets that you intend to use as `og:image` property in your views. This is django-meta's counterpart of the Django's `STATIC_URL` setting. In fact, Django's `STATIC_URL` setting is a fallback if you do not specify this setting, so make sure either one is configured. Default is to use the `STATIC_URL` setting.

Note that you must add the trailing slash when specifying the URL. Even if you do not intend to use the `og:image` property, you need to define either this setting or the `STATIC_URL` setting or an attribute error will be raised.

1.6.8 META_USE_OG_PROPERTIES

This setting tells django-meta whether to render the OpenGraph properties. Default is `False`.

1.6.9 META_USE_TWITTER_PROPERTIES

This setting tells django-meta whether to render the Twitter properties. Default is `False`.

1.6.10 META_USE_SCHEMAORG_PROPERTIES

This setting tells django-meta whether to render the Schema.org properties. Default is `False`.

1.6.11 META_USE_TITLE_TAG

This setting tells django-meta whether to render the `<title></title>` tag. Default is `False`.

1.6.12 META_USE_SITES

This setting tells django-meta to derive the site's domain using the Django's sites contrib app. If you enable this setting, the *META_SITE_DOMAIN* is not used at all. Default is `False`.

1.6.13 META_OG_NAMESPACES

Use this setting to add a list of additional OpenGraph namespaces to be declared in the `<head>` tag.

1.6.14 META_OG_SECURE_URL_ITEMS

Use this setting to customize the list of additional OpenGraph properties for which the `:secure_url` suffix is added if the URL value is a `https` URL.

1.6.15 Other settings

The following settings are available to set a default value to the corresponding attribute for both *View support* and *Model support*

- image: `META_DEFAULT_IMAGE` (must be an absolute URL, ignores *META_IMAGE_URL*)
- object_type: `META_SITE_TYPE` (default: first `META_OBJECT_TYPES`)
- og_type: `META_FB_TYPE` (default: first `META_FB_TYPES`)
- og_app_id: `META_FB_APPID` (default: blank)
- og_profile_id: `META_FB_PROFILE_ID` (default: blank)
- fb_pages: `META_FB_PAGES` (default: blank)
- og_publisher: `META_FB_PUBLISHER` (default: blank)
- og_author_url: `META_FB_AUTHOR_URL` (default: blank)
- twitter_type: `META_TWITTER_TYPE` (default: first `META_TWITTER_TYPES`)

- twitter_site: `META_TWITTER_SITE` (default: blank)
- twitter_author: `META_TWITTER_AUTHOR` (default: blank)
- schemaorg_type: `META_SCHEMAORG_TYPE` (default: first `META_SCHEMAORG_TYPE`)

1.7 Rendering meta tags

To render the meta tags, simply add the `meta` dictionary/object to the template context, and add this inside the <head> tags:

```
{% include 'meta/meta.html' %}
```

The partial template will not output anything if the context dictionary does not contain a `meta` object, so you can safely include it in your base template.

Additionally, if you want to use facebook or a custom namespace, you should include them in the <head> tag, as follow:

```
{% load meta %}
...
<head {% meta_namespaces %} >
```

This will take care of rendering OpenGraph namespaces in the <head prefix="...">.

If you enabled Schema.org support and you want to mark the whole page as Schema.org object, add the following templatetag to the <html> tag:

```
{% load meta %}
...
<html {% meta_namespaces_schemaorg %}>
```

For compatibility with 1.0 and previous version you can keep the sekizai version of the above:

```
{% load sekizai_tags meta %}
...
<html {% render_block 'html_extra' %}>
```

1.8 Adding custom tags / properties

Both *models* and *views* support adding custom tags and properties by extending the respective python classes.

They both rely on the following attributes:

- `extra_props`: use this to add new meta tags using the `name` attribute as key. Example:

  ```
  <meta name="designer" content="Pablo Picasso">
  ```

- `extra_custom_props`: use this to add new meta tags using a custom attribute as key. Example:

  ```
  <meta property="og:type" content="music.song" />
  <meta property="music:duration" content="3" />
  ```

Using this approach, you won't need to change the `meta.html` template to include your custom tags.

See below concrete implementation examples.

1.8.1 Format

extra_props

A dictionary of extra optional properties:

```
{
    'foo': 'bar',
    'key': 'value'
}
```

```
<meta name="foo" content="bar">
<meta name="key" content="value">
```

extra_custom_props

A list of tuples for rendering custom extra properties:

```
[
    ('key', 'foo', 'bar')
    ('property', 'name', 'value')
]
```

```
<meta name="foo" content="bar">
<meta property="name" content="value">
```

1.8.2 Views

To add custom tags / properties you can follow the same specifications detailed in *Using the view*.

- Pass the values to the `Meta` object (see *Meta object*):

```
meta = Meta(
   ...
    extra_props={
        'designer': 'Pablo Picasso',
    },
    extra_custom_props=[
        ('property', 'og:type', 'music.song'),
        ('property', 'music:duration', '3')
    ]
    ...
)
```

- add as attributes to the view using *meta.views.MetadataMixin* (see *View mixin*):

```
class MyView(MetadataMixin, ListView):
    ...
    extra_props = {
        'designer': 'Pablo Picasso',
    }
```

(continues on next page)

(continued from previous page)

```
    extra_custom_props = [
        ('property', 'og:type', 'music.song'),
        ('property', 'music:duration', '3')
    ]
    ...
```

1.8.3 Models

For models they need to be added to the `_metadata` attribute as per the other properties (see *Usage*).

As the other properties you can both provide the static value (see `extra_props` below, or the name of a callable which will return the value at runtime (see `extra_custom_props`).

```
class Post(ModelMeta, models.Model):
    ...
    _metadata = {
        ...
        'extra_props': {
            'designer': 'Pablo Picasso',
        },
        'extra_custom_props': 'get_custom_props'
    }
    ...
    def get_custom_props(self):
        return [
            ('property', 'og:type', 'music.song'),
            ('property', 'music:duration', '3')
        ]
    ...
```

1.9 Package documentation

class meta.views.**FullUrlMixin**

Provides a few convenience methods to retrieve the full URL (which includes protocol and domain) of an object.

If possible, `django.http.request.HttpRequest.build_absolute_uri()` is used

_get_full_url(*url*)

Build the full URL (protocol and domain included) for the URL given as argument

Parameters
url – absolute (domain-less) URL

Returns
full url

get_domain()

Discover the current website domain

`django.contrib.sites.models.Site` and *META_SITE_DOMAIN* (in this order) are used

Returns
domain URL

get_protocol()

Discover the current website protocol from *META_SITE_PROTOCOL*

Returns
http or https depending on *META_SITE_PROTOCOL*

class meta.views.**Meta**(***kwargs*)

Helper for building context meta object

_image = None

_image_object = None

_keywords = []

_normalize_media_url(*url*)

_obj = None

Linked `ModelMeta` instance (if Meta is generated from a ModelMeta object)

_schema = {}

Base schema.org types definition.

It's a dictionary containing all the schema.org properties for the described objects.

See *a sample implementation*.

_url = None

as_json_ld()

Convert the schema to json-ld

Returns
json

get_full_url(*url*)

property image

property image_object

property keywords

request = None

property schema

Schema.org object description.

Items in the schema are converted in a format suitable of json encoding at this stage:

- instances of `Meta` as their schema
- dates as isoformat
- iterables and dicts are processed depth-first to process their items

If no type is set `schemaorg_type` is used

Returns
dict

property url

class meta.views.**MetadataMixin**(***kwargs*)

Django CBV mixin to prepare metadata for the view context

context_meta_name = 'meta'

custom_namespace = None

description = None

extra_custom_props = None

extra_props = None

facebook_app_id = None

get_context_data(***kwargs*)

get_domain()

Discover the current website domain

`django.contrib.sites.models.Site` and *META_SITE_DOMAIN* (in this order) are used

Returns
domain URL

get_meta(*context=None*)

get_meta_class()

get_meta_custom_namespace(*context=None*)

get_meta_description(*context=None*)

get_meta_extra_custom_props(*context=None*)

get_meta_extra_props(*context=None*)

get_meta_facebook_app_id(*context=None*)

get_meta_image(*context=None*)

get_meta_image_object(*context=None*)

get_meta_keywords(*context=None*)

get_meta_locale(*context=None*)

get_meta_object_type(*context=None*)

get_meta_og_title(*context=None*)

get_meta_schemaorg_description(*context=None*)

get_meta_schemaorg_title(*context=None*)

get_meta_schemaorg_type(*context=None*)

get_meta_site_name(*context=None*)

get_meta_title(*context=None*)

get_meta_twitter_card(*context=None*)

get_meta_twitter_creator(*context=None*)

get_meta_twitter_site(*context=None*)

get_meta_twitter_title(*context=None*)

get_meta_twitter_type(*context=None*)

get_meta_url(*context=None*)

get_protocol()

Discover the current website protocol from *META_SITE_PROTOCOL*

Returns
http or https depending on *META_SITE_PROTOCOL*

get_schema(*context=None*)

The generic API to retrieve the full schema.org structure for the view.

By default it returns the `schema`. You can reimplement this method to build the schema.org structure at runtime. See *a sample implementation*.

Parameters
context – view context

Returns
dictionary

get_schema_property(*schema_type*, *property*, *context=None*)

The generic API to retrieve the attribute value for a generic schema type

This is just a stub that **must** be implemented

Parameters

- **schema_type** – name of the schema type
- **property** – name of the property
- **context** – view context

Returns
property value

image = None

image_object = None

keywords = []

locale = None

meta_class

alias of `Meta`

object_type = None

og_title = None

schema = {}

schemaorg_description = None

schemaorg_title = None

schemaorg_type = None

site_name = None

title = None

property twitter_card

twitter_creator = None

twitter_site = None

twitter_title = None

twitter_type = None

url = None

use_og = False

use_sites = False

use_title_tag = False

class meta.models.**ModelMeta**

Meta information mixin.

_metadata = {}

Metadata configuration dictionary

_metadata dict values can be:

- name of object method taking the field name as parameter
- name of object method taking no parameters
- name of object attribute
- name of callable taking the field name as parameter
- name of callable taking no parameters
- literal value

They are checked in the given order: the first that matches is returned.

Callable must be available in the module (i.e.: imported if not defined in the module itself)

_schema = {}

schema.org properties dictionary

_metadata dict values can be:

- name of object method taking the field name as parameter
- name of object method taking no parameters
- name of object attribute
- name of callable taking the field name as parameter

- name of callable taking no parameters
- literal value

They are checked in the given order: the first that matches is returned.

Callable must be available in the module (i.e.: imported if not defined in the module itself)

If the resulting value is a *ModelMeta* or *Meta* instance its schema is set in the schema.org dataset.

See *a sample implementation*.

as_meta(*request=None*)

Populates the *Meta* object with values from *_metadata*

Parameters
request – optional request object. Used to build the correct URI for linked objects

Returns
Meta object

build_absolute_uri(*url*)

Return the full url for the provided url argument

get_author()

Retrieve the author object. This is meant to be overridden in the model to return the actual author instance (e.g.: the user object).

get_author_name()

Sample method to return the author full name

get_author_schemaorg()

Sample method to return the author Schema.org URL

get_author_twitter()

Sample method to return the author twitter account

get_author_url()

Sample method to return the author facebook URL

get_meta(*request=None*)

Retrieve the meta data configuration

get_meta_protocol()

Current http protocol

get_request()

Retrieve request from current instance

mainEntityOfPage()

schema

Schema.org object description

Returns
dict

meta.templatetags.meta.**custom_meta**(*attr*, *name*, *content*)

Generates a custom meta tag:

<meta {attr}=”{name}” content=”{content}”>

Parameters

- **attr** – meta attribute name
- **name** – meta name
- **content** – content value

meta.templatetags.meta.**custom_meta_extras**(*extra_custom_props*)

Generates the markup for a list of custom meta tags

Each tuple is passed to :py:func:custom_meta to generate the markup

Parameters

extra_custom_props – list of tuple of additional meta tags

meta.templatetags.meta.**facebook_prop**(*name*, *value*)

Generic Facebook property

Parameters

- **name** – property name (without 'fb:' namespace)
- **value** – property value

meta.templatetags.meta.**generic_prop**(*namespace*, *name*, *value*)

Generic property setter that allows to create custom namespaced meta e.g.: fb:profile_id.

meta.templatetags.meta.**googleplus_html_scope**(*value*)

Legacy Google+ scope

meta.templatetags.meta.**googleplus_prop**(*name*, *value*)

Legacy Google+ property

meta.templatetags.meta.**googleplus_scope**(*value*)

Legacy Google+ scope

meta.templatetags.meta.**meta**(*name*, *content*)

Generates a meta tag according to the following markup:

<meta name="{name}" content="{content}">

Parameters

- **name** – meta name
- **content** – content value

meta.templatetags.meta.**meta_extras**(*extra_props*)

Generates the markup for a list of meta tags

Each key,value paur is passed to :py:func:meta to generate the markup

Parameters

extra_props – dictionary of additional meta tags

meta.templatetags.meta.**meta_list**(*name*, *lst*)

Renders in a single meta a list of values (e.g.: keywords list)

Parameters

- **name** – meta name
- **lst** – values

meta.templatetags.meta.**meta_namespaces**(*context*)

Include OG namespaces. To be used in the <head> tag.

meta.templatetags.meta.**meta_namespaces_gplus**(*context*)

Legacy Google+ attributes.

meta.templatetags.meta.**meta_namespaces_schemaorg**(*context*)

Include Schema.org attributes. To be used in the <html> or <body> tag.

meta.templatetags.meta.**og_prop**(*name*, *value*)

Generic OpenGraph property

Parameters

- **name** – property name (without 'og:' namespace)
- **value** – property value

meta.templatetags.meta.**schemaorg_html_scope**(*value*)

This is meant to be used as attribute to html / body or other tags to define schema.org type

Parameters

value – declared scope

meta.templatetags.meta.**schemaorg_prop**(*name*, *value*)

Generic Schema.org property

Parameters

- **name** – property name
- **value** – property value

meta.templatetags.meta.**schemaorg_scope**(*value*)

Alias for googleplus_html_scope

Parameters

value – declared scope

meta.templatetags.meta.**title_prop**(*value*)

Title tag

Parameters

value – title value

meta.templatetags.meta.**twitter_prop**(*name*, *value*)

Generic Twitter property

Parameters

- **name** – property name (without 'twitter:' namespace)
- **value** – property value

1.10 Development & community

django meta is an open-source project.

You don't need to be an expert developer to make a valuable contribution - all you need is a little knowledge, and a willingness to follow the contribution guidelines.

1.10.1 Nephila

django meta is maintained by Iacopo Spalletti at Nephila and is released under a BSD License.

Nephila is an active supporter of Django, django CMS and its community. django meta is intended to help make it easier for developers in the Django ecosystem to work effectively and create high quality applications.

1.11 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

Please read the instructions here to start contributing to *django-meta*.

1.12 History

1.12.1 2.4.2 (2024-02-05)

Features

- Add uz translations (#198)

1.12.2 2.4.1 (2023-12-13)

Features

- Remove useless "else" statements. (#182)
- Switch to Coveralls Github action (#188)

Bugfixes

- Refactor FullUrlMixin get_domain to handle django.contrib.sites not in INSTALLED_APPS (#192)

1.12.3 2.4.0 (2023-09-25)

Features

- Add schema.org support (#76)
- Refactor settings to make override_settings in tests more consistent (#167)
- Migrate to bump-my-version (#173)

1.12.4 2.3.0 (2023-08-10)

Bugfixes

- Fix schemaorg_description not being in Meta class (#127)
- Fix schema.org protocol to be https (#152)
- Fix request set order in Meta.__init__ (#155)

1.12.5 2.2.0 (2023-04-18)

Features

- Move to ruff (#138)
- Add support for Django 4.2 (#144)

1.12.6 2.1.0 (2022-07-28)

Bugfixes

- Changes imports from ugettext_lazy to gettext_lazy to fix deprecation warning (#130)
- Get correct setting META_USE_SITES in build_absolute_uri model method (#133)
- Update tox environments and github actions (#135)

1.12.7 2.0.0 (2020-11-14)

Features

- Drop Python 2 (#118)
- Drop Django<2.2 (#118)
- Add Django 3.1 (#118)
- Update tooling (#118)
- Port to github-actions (#118)
- Remove G+ support - Replace with Schema.org (#108)
- Add support for image object (#114)

Bugfixes

- Switch request handling to thread locals (#115)

1.12.8 1.7.0 (2020-07-07)

- Fixed support for secure_url
- Normalized twitter_card / twitter_type attributes

1.12.9 1.6.1 (2020-01-16)

- Added explicit dependency on six
- Added python 3.8

1.12.10 1.6.0 (2019-12-22)

- Added Django 3.0 support
- Moved to django-app-helper
- Improved documentation regarding extra / custom props

1.12.11 1.5.2 (2019-07-02)

- Added image size for facebook sharing

1.12.12 1.5.1 (2019-04-11)

- Fixed error if the property referenced in _metadata returns False

1.12.13 1.5.0 (2019-03-23)

- Added support for Django 2.1 and 2.2
- Added support for Python 3.7
- Dropped support for Django < 1.11
- Dropped support for Python 3.4
- Fixed support for og:image:secure_url
- Fixed minor documentation error
- Added support for service-specific titles

1.12.14 1.4.1 (2018-01-21)

- Added Django 2.0 support
- Fixed RTD builds
- Fixed MetadataMixin.use_use_title_tag typo
- Add request to Meta arguments

1.12.15 1.4.0 (2017-08-12)

- Add Django 1.11 support
- Drop python 2.6/ Django<1.8
- Wrap meta.html content in spaceless templatetag to suppress redundant newlines
- Fix issue in Django 1.10

1.12.16 1.3.2 (2016-10-26)

- Fix error if custom_meta_extras is empty
- Fix twitter properties
- Fix error with META_DEFAULT_IMAGE path

1.12.17 1.3.1 (2016-08-01)

- Add support for G+ publisher tag

1.12.18 1.3 (2016-06-06)

- Added support for fb_pages attribute
- Properly implement META_DEFAULT_IMAGE for view-based mixins
- Fixed error in facebook_prop templatetag
- Removed dependency of sites framework

1.12.19 1.2 (2016-04-09)

- Fix issue when emulating sekizai

1.12.20 1.1 (2016-04-08)

- Sekizai is not required anymore

1.12.21 1.0 (2016-03-29)

- Merge with django-meta-mixin
- Reorganized documentation
- Remove deprecated `make_full_url` method
- Add _retrieve_data interface for generic attribute data generation

1.12.22 0.3.2 (2016-02-09)

- Use autoescape off in template for Django 1.9

1.12.23 0.3.1 (2015-06-27)

- Bump for re-upload

1.12.24 0.3.0 (2015-06-27)

- Add support for more twitter attributes
- Add support for more facebook attributes
- Official support for Django 1.4->1.8
- Official support for Python 2.6, 2.7, 3.2, 3.3, 3.4

1.12.25 0.2.1 (2014-12-15)

- Add support for more attributes
- Add templatetag to handle generic attributes

1.12.26 0.2.0 (2014-05-28)

- Code cleanup
- Change maintainership information
- Official Python 3 support

1.12.27 0.1.0 (2014-01-20)

- Support for Twitter meta data (leifdenby)
- Fixes to OpenGraph tags (leifdenby)
- Support Google Plus tags (Iacopo Spalletti)

1.12.28 0.0.3 (2013-11-12)

- Keywords are now order-preserving
- Keywords are no longer a set(), but a normal list

1.12.29 0.0.2 (2013-04-12)

- Fixed keywords not being included in metadata
- Fixed get_meta_class not being used in the mixin

1.12.30 0.0.1 (2013-04-04)

- Initial version

CHAPTER

TWO

APPS USING DJANGO-META / EXTENSIONS

- djangocms-blog: https://github.com/nephila/djangocms-blog
- djangocms-page-meta: https://github.com/nephila/djangocms-page-meta
- django-knocker: https://github.com/nephila/django-knocker
- wagtail-metadata-mixin: https://github.com/bashu/wagtail-metadata-mixin

Open a pull request to add yours here

2.1 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

O

R

S

T

U