django-missing Documentation

Release 0.2.1

wlan slovenija

Jul 04, 2017

Contents

This Django application bundles some common and useful features which have not (yet) found a way into Django itself.

You are invited to contribute your own features so that together we create a powerful and good library of Django additions. Because sometimes you cannot imagine an useful feature before somebody shows it to you. So it is useful to have all of them collected together.

CHAPTER 1

Contents

Middleware

class missing.middleware.**ForceAdminLanguage**

Middleware which forces language in Django admin to ADMIN_LANGUAGE_CODE setting value.

Useful when not wanting that Django content language interferes with admin language, especially when admin interface is not translated fully in all languages content is, or when error messages in admin interface are hard to debug because of a rare language they are displayed in.

Should be added to MIDDLEWARE_CLASSES after LocaleMiddleware middleware:

```
MIDDLEWARE_CLASSES = (
    ...
    django.middleware.locale.LocaleMiddleware,
    missing.middleware.ForceAdminLanguage,
    ...
)
```

Template Tags and Filters

context_tags module

You need to add missing.templatetags.context_tags to your builtins option for the DjangoTemplates backend to be able to use these tags immediately after extends tag.

missing.templatetags.context_tags.**contextblock**(*parser*, *token*)

A special block tag which does not render anything but can be used to modify a template context.

The tag is rendered first thus modifying context before other blocks are rendered. A tag in an extending template is rendered after parent tags, allowing you to override template context in child templates.

The tag has to be the first tag, immediately after the `extends` tag. You have to define a empty context block tag at the very start of your base template.

Example usage, in your base template:

```
{% contextblock %}{% endcontextblock %}<html>
    <body>
        <head>
            <title>{{ title }}</title>
        </head>
        <body>
            <h1>{{ title }}</h1>
            <p><a href="{{ homepage }}">{{ title }}</a></p>
        </body>
    </body>
</html>
```

In your extending template:

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

{% contextblock %}
    {% load future i18n %}
    {% setcontext as title %}{% blocktrans %}{{ username }}'s blog{%
↪endblocktrans %}{% endsetcontext %}
    {% url "homepage" as homepage %}
{% endcontextblock %}
```

`missing.templatetags.context_tags.`**`setcontext`**(*parser*, *token*)

Sets (updates) current template context with the rendered output of the block inside tags.

This is useful when some template tag does not support storing its output in the context itself or we need some complex content (like language, user or URL dependent content) multiple times.

Example usage:

```
{% setcontext as varname %}
    {% complextag %}
{% endsetcontext %}
```

`forloop_tags` module

Use `{% load forloop_tags %}` in your template to load this module.

`missing.templatetags.forloop_tags.`**`css_classes`**(*context*, *first*, *last*, *odd*, *even*)

Maps loop variables of `for` tag to CSS classes string.

Takes names of first, last, odd, even CSS classes, respectively

Example usage:

```
{% css_classes "first" "last" "odd" "even" %}
```

`html_tags` module

Use `{% load html_tags %}` in your template to load this module.

missing.templatetags.html_tags.**anchorify**(*anchor*)

Filter which converts to a string suitable for use as an anchor id on a HTML element.

This is useful when you want anchor id on a heading to match heading content, which can be an arbitrary string.

Example usage:

```
<h1 id="{{ _("My Blog")|anchorify }}">{% trans "My Blog" %}</h1>
```

The result would be:

```
<h1 id="my-blog">My Blog</h1>
```

missing.templatetags.html_tags.**heading**(*context*, *level*, *content*, *classes=''*)

Renders heading with unique anchor id using a template.

Tag assures that each anchor id is unique inside the whole rendered template where it is used. Of course, only for headings created with the tag.

Heading level will be adjusted based on base heading level set by *set_base_heading_level()* or `base_heading_level` context variable. This is useful if you have some static main `<h1>` with a site name and you want other headings to have a higher level automatically, but you want to reuse the same template you use can independently, without site name heading. Or if you include same template which uses this tag at various places where different heading level is needed based on the existing heading nesting. By default base heading level is 0, so no adjustment will be made, so example below will make a `<h1>` heading.

Optionally you can pass CSS classes string which will be passed through to the heading template. It uses `heading.html` or `missing/heading.html` template.

Example usage:

```
{% heading 1 _("My Blog") %}
```

The result would be:

```
<h1 id="my-blog" class="heading ">My Blog</h1>
```

missing.templatetags.html_tags.**set_base_heading_level**(*context*, *level*, *top_level=False*)

Set base heading level to a given numeric level to adjust heading levels of headings created by the *heading()* tag.

You can also set base heading level by setting `base_heading_level` context variable. For example, by using built-in `with` tag.

If you set `top_level` to `True`, base heading level will be set at the top context level for the whole template. This is useful if you want to set base heading level for the whole template, but you are using the tag somewhere nested in blocks and includes. It will set base heading level only at the top context level so if you set heading level explicitly at some other context levels as well they will still take precedence.

Example usage:

```
{% set_base_heading_level 1 %}
{% heading 1 _("My Blog") %}
```

The result would be:

```
<h2 id="my-blog" class="heading ">My Blog</h2>
```

`lang_tags` module

Use `{% load lang_tags %}` in your template to load this module.

`missing.templatetags.lang_tags.`**`translate`**(*string*, *lang_code*)
Translates given string to the specified language.

This is useful for text you need in some other language than the current language. For example, for links inviting users to switch to their language.

Example usage:

```
{% translate "Do you understand this?" "de" %}
```

`list_tags` module

Use `{% load list_tags %}` in your template to load this module.

`missing.templatetags.list_tags.`**`divide_list`**(*value*, *count*)
Divides input list into the given number of sublists.

Last sublist can be shorter if input list length is not a multiplier of the given number of sublists.

Example usage:

```
<tr>
{% for column in objects|divide_list:"2" %}
    <td><ul>
    {% for obj in column %}
        <li>{{ obj }}</li>
    {% endfor %}
    </ul></td>
{% endfor %}
</tr>
```

`missing.templatetags.list_tags.`**`split_list`**(*value*, *length*)
Splits input list into sublists of the given length.

Last sublist can be shorter if input list length is not a multiplier of the given length.

Example usage:

```
{% for group in objects|split_list:"4" %}
    <tr>
    {% for obj in group %}
        <td>{{ obj }}</td>
    {% endfor %}
    </tr>
{% endfor %}
```

`string_tags` module

Use `{% load string_tags %}` in your template to load this module.

`missing.templatetags.string_tags.`**`count`**(*value*, *arg*)
Returns the number of non-overlapping occurrences of an argument substring in the given string.

missing.templatetags.string_tags.**ensure_sentence**(**args*, ***kwargs*)
Ensures that string ends with dot if it does not already end with some punctuation.

missing.templatetags.string_tags.**startswith**(*value*, *arg*)
Returns True if the given string starts with an argument prefix, otherwise returns False.

`url_tags` module

Use `{% load url_tags %}` in your template to load this module.

missing.templatetags.url_tags.**active_url**(*context*, *urls*, *class_name='active'*)
Returns `class_name` (default `active`) if any of given `urls` are real prefixes of the current request path.

Useful when you want to highlight links to the current section of the site. For example, in menu entries.

Example usage:

```
{% active_url "/test/" %}
```

missing.templatetags.url_tags.**fullurl**(*parser*, *token*)
Builds an absolute (full) URL from the given location and the variables available in the request.

If no location is specified, the absolute (full) URL is built on `django.http.HttpRequest.get_full_path()`.

It is a wrapper around `django.http.HttpRequest.build_absolute_uri()`. It requires `request` to be available in the template context (for example, by using `django.core.context_processors.request` context processor).

Example usage:

```
{% url "view_name" as the_url %}
{% fullurl the_url %}
```

missing.templatetags.url_tags.**slugify2**(**args*, ***kwargs*)
Normalizes string, converts to lowercase, removes non-alpha characters, and converts spaces to hyphens.

It is similar to built-in `slugify` but it also handles special characters in variety of languages so that they are not simply removed but properly transliterated/downcoded.

missing.templatetags.url_tags.**urltemplate**(*context*, *viewname*, **args*, ***kwargs*)
Creates URI template in a similar way to how `url` tags work but leaving parts of a URI to be filled in by a client. See **RFC 6570** for more information.

Names of parts are taken from named groups in URL regex pattern used for the view, or as a part's sequence number (zero-based) for unnamed groups. You can pre-fill some parts by specifying them as additional arguments to the tag.

> **Warning:** Tag cannot check if pre-fill values specified will really match back the URL regex pattern, so make sure yourself that they do.

Example usage:

```
{% with variable="42" %}
    {% urltemplate "view_name" arg1="value" arg2=variable %}
{% endwith %}
```

If URL pattern would be defined like:

```
url(r'^some/view/(?P<arg1>.*)/(?P<arg2>.*)/(?P<param>.*)/$', some_view, name=
↪'view_name'),
```

The result would be:

```
/some/view/value/42/{param}/
```

Views

class missing.views.**EnsureCsrfCookieMixin**
Mixin for Django class-based views which forces a view to send the CSRF cookie.

This is useful when using Ajax-based sites which do not have an HTML form with a csrf_token that would cause the required CSRF cookie to be sent.

missing.views.**bad_request_view**(*request*, *exception=None*)
Displays 400 bad request page.

It is similar to the Django built-in django.views.defaults.permission_denied view, but always uses a template and a request context. You can configure Django to use this view by adding to urls.py:

```
handler400 = 'missing.views.bad_request_view'
```

Template should not require a CSRF token.

missing.views.**forbidden_view**(*request*, *exception=None*, *reason=''*)
Displays 403 forbidden page. For example, when request fails CSRF protection.

It is similar to a merged Django built-in django.views.defaults.permission_denied and django.views.csrf.csrf_failure views, but always uses a template and a request context. You can configure Django to use this view by adding to urls.py:

```
handler403 = 'missing.views.forbidden_view'
```

and to settings.py:

```
CSRF_FAILURE_VIEW = 'missing.views.forbidden_view'
```

Template should not require a CSRF token.

Debugging

Safer ExceptionReporterFilter

class missing.debug.**SafeExceptionReporterFilter**
Safe exception reporter filter which also filters request environment (META) and cookies (COOKIES) so that it is safer to share the report publicly.

This is useful to not display passwords and other sensitive data passed to Django through its process environment.

Furthermore, it configures Django to additionally clean settings with URL, CSRF, COOKIE, csrftoken, csrfmiddlewaretoken, and sessionid in keys.

To install it, configure Django to:

```
DEFAULT_EXCEPTION_REPORTER_FILTER = 'missing.debug.SafeExceptionReporterFilter'
```

and import `missing.debug` somewhere in your code, for example, in `urls.py` of your project.

Non-silent NoReverseMatch

`NoReverseMatch` is by default a silent exception in variables, its output replaced by `TEMPLATE_STRING_IF_INVALID` setting. Sometimes you want a bit more loud expression of mismatched URL reversing, so you can set `URL_RESOLVERS_DEBUG` to `True` to normally raise an exception. Only active when `DEBUG` is set to `True` as well.

Timezone

`missing.timezone.`**`to_date`**(*value*)

Function which knows how to convert timezone-aware `datetime` objects to `date` objects, according to guidelines from Django documentation.

JavaScript

Automatic Slug Generation

Django admin and other Django applications can have slug fields which are automatically updated/generated in user's browser using JavaScript. Django bundles JavaScript code necessary for this but it behaves differently than built-in `slugify` template filter in Python. For this reason django-missing provides JavaScript code with equal functionality, implemented directly in JavaScript. It comes in two flavors:

`slugify` equivalent

A JavaScript equivalent to built-in `slugify` template filter. You can load by adding something like this in your page (or Django admin) `<head>` section (in template):

```
<script type="text/javascript" src="{{ STATIC_URL }}missing/n11ndata.js"></script>
<script type="text/javascript" src="{{ STATIC_URL }}missing/n11n.js"></script>
<script type="text/javascript" src="{{ STATIC_URL }}missing/urlify.js"></script>
```

Of course above mentioned files should be published by your Django site installation.

`slugify2` equivalent

If you want to use improved *`slugify2()`* template filter in Python, you can also use its equivalent in JavaScript:

```
<script type="text/javascript" src="{{ STATIC_URL }}missing/n11ndata.js"></script>
<script type="text/javascript" src="{{ STATIC_URL }}missing/n11n.js"></script>
<script type="text/javascript" src="{{ STATIC_URL }}missing/urlify2.js"></script>
```

Datetime Formatting

Once you load internationalization in JavaScript code, Django provides `get_format` function to access configured datetime and other formats but it is lacking function to format JavaScript `Date` objects according to those formats. By loading:

```
<script type="text/javascript" src="{{ STATIC_URL }}missing/date.js"></script>
```

JavaScript `Date` prototype is extended with `strfdate` method:

```
new Date().strfdate(get_format('DATETIME_FORMAT'))
```

Note, to format datetime input formats (those using `%` for placeholders) Django admin provides limited support through its `strftime` method added to JavaScript `Date` prototype when loading:

```
<script type="text/javascript" src="{{ STATIC_URL }}admin/js/core.js"></script>
```

and use, for example, as:

```
new Date().strftime(get_format('DATE_INPUT_FORMATS')[0])
```

Relative datetime

JavaScript implementations of `timesince`, `timeuntil`, and `naturaltime` Django filters are also available as extensions to JavaScript `Date` prototype by loading:

```
<script type="text/javascript" src="{{ STATIC_URL }}missing/humanize.js"></script>
```

Additionally, `updatingNaturaltime` method is provided which behaves similarly to `naturaltime` method but it takes a DOM element or jQuery selector as an optional argument and makes sure it is updated as time progresses.

CHAPTER 2

Source Code and Issue Tracker

GitHub is used for development, so source code and issue tracker is found there. Feel free to fork and contribute with pull requests, or open issues you might have. Ideas for improvements are welcome, too.

CHAPTER 3

Indices and Tables

- genindex
- modindex
- search

Python Module Index

m

Index