django-tabination Documentation

Release 0.4.0

Danilo Bargen

November 27, 2016

Contents

django-tabination is a library that enables you to easily build your own tab navigation based on class based views.

The main idea behind this library is that the properties of the tabs are defined inside the view and aren't stored in the database. The database based approach (which is used for example by django-sitetree or django-treenav, often based on django-mptt) is great for CMS-like projects with users editing the pages directly via the admin, but it causes many problems when the pages are mainly coded directly in the views because the navigation is then not tracked by your version control system and can be off-sync / inconsistent between different versions or systems.

There are also projects that provide a set of template tags to mark a page as active, which can then be used to render the navigation template accordingly (e.g. django-tabs). But that solution is very limited and not as flexible as *django-tabination*.

django-tabination allows you to create tabs directly in your class based views by settings some specific attributes. This can be simplified even further by creating a common base class for all your tab views that handles all the logic necessary to build a dynamically configured tab navigation.

Features include conditional displaying/hiding of a tab, translation of the tab labels, tab hierarchies to build multi-level navigations and more.

CHAPTER 1

Table of Contents

1.1 Installation and Configuration

There are several ways to install *django-tabination*, either by using a package manager like pip or by manually downloading and installing a copy of the library.

1.1.1 Installing

The recommended way to install *django-tabination* is directly from pypi using pip:

```
pip install django-tabination
```

If you prefer not to use an automated package installer, you can download a copy of *django-tabination* and install it manually. To install it, navigate to the directory containing `setup.py` on your console and type:

```
python setup.py install
```

1.1.2 Configuration

Currently there is no further configuration needed to use *django-tabination*. Take a look at the Usage docs to see how to implement your tabs.

1.1.3 Source Code

The source code of *django-tabination* is licensed under the LGPLv3 license and can be forked on GitHub.

1.2 Usage

django-tabination is a library that enables you to easily build your own tab navigation templates by extending the `views.TabView` base class.

The library is strongly based on the class based views that Django has introduced with version 1.3. You cannot use this library if your project is using function based views.

1.2.1 Creating tab views

For a working custom tab view, the following things are requried:

- You need to extend the `views.TabView` base class
- You need to add the class attribute `_is_tab = True` to your view

- You need to specify the `tab_group`.
- Each tab needs a `tab_id`.
- In order for the tab to be visible in your navigation, you need to set a `tab_label`.
- You need to define a `template_name`.

Note: The `_is_tab` attribute is needed for the class to be tracked by a tracking metaclass. Therefore it needs to be present when the classes are parsed by the Python interpreter and cannot be added later, e.g. with a decorator.

Getting started

The base class resides in `tabination.views`. Import it like this:

```
from tabination.views import TabView
```

This is a very simple example tab:

```
class SpamTab(TabView):
    _is_tab = True
    tab_id = 'spam'
    tab_group = 'main_navigation'
    tab_label = 'Spam'
    template_name = 'tabs/spam_tab.html'
```

Now your page will be rendered using the template *tabs/spam_tab.html*, because `views.TabView` extends Django's generic *TemplateView*.

If you want, you can also use other generic view mixins (or any other custom mixins) to provide additional functionality. A good example would be the *SingleObjectMixin*:

```
from django.views.generic.detail import SingleObjectMixin

class SpamTab(SingleObjectMixin, TabView):
    _is_tab = True
    tab_id = 'spam'
    tab_group = 'main_navigation'
    tab_label = 'Spam'
    template_name = 'tabs/spam_tab.html'
    model = models.SpamCan
```

Now the *SpamCan* object with a primary key provided from your URL definition will be passed on to your template as `object` (see SingleObjectMixin documentation).

Warning: As of Django 1.4, above example does not work due to a bug in the class based views implementation (`get_context_data` in the generic mixins does not call `super()`). This is fixed in Django 1.5 (see Ticket #16074). If you're still using Django 1.4 you can either use generic mixins that don't affect `get_context_data`, manually call `TabView.get_context_data(self, **kwargs)` from your tab code or create your own mixins. See the next section for an example.

You can do everything with your TabView that you can do with normal class based views. The only things that you need to bear in mind is that `views.TabView` always needs to be the base class (on the right side of the parentheses). It may be overloaded using mixins but cannot be combined with other views that override `get_context_data`.

Customizing your tab view

You can further customize your tab view by overloading the `views.TabView`'s class attributes with your own class- or instance attributes or properties (if logic is required).

For available attributes, see `views.TabView` documentation. You can also create your own attributes, as long as they're used in your template.

Keep in mind that if the tab you're working with is not the currently loaded tab, it is just an instance of the tab that has not passed through the dispatching functions. In case you need some variables that you get only by dispatching the request (e.g. `self.kwargs`), you can use the special attribute `self.current_tab` to gain access to the currently loaded tab. See also section *Accessing request data*.

Here is an example of a more sophisticated tab view hierarchy:

```
from django.contrib.auth.decorators import login_required
from django.utils import decorators
from django.utils.translation import ugettext as _

from tabination.views import TabView


class MainNavigationBaseTab(TabView):
    """Base class for all main navigation tabs."""
    tab_group = 'main_navigation'
    tab_classes = ['main-navigation-tab']

    def get_context_data(self, **kwargs):
        context = super(MainNavigationBaseTab, self).get_context_data(**kwargs)
        context['spam'] = 'ham'
        return context

    @property
    def tab_classes(self):
        """If user is logged in, set ``logged_in_only`` class."""
        classes = super(MainNavigationBaseTab, self).tab_classes[:]
        if self.current_tab.request.user.is_authenticated():
            classes += ['logged_in_only']
        return classes


class SpamTab(MainNavigationBaseTab):
    """A simple TabView."""
    _is_tab = True
    tab_id = 'spam'
    tab_label = _('Spam')
    template_name = 'spam_tab.html'


class HamTab(MainNavigationBaseTab):
    """TabView is only visible after authentication."""
    _is_tab = True
    tab_id = 'ham'
    tab_label = _('Ham')
    tab_rel = 'nofollow,noindex'
    template_name = 'ham_tab.html'

    @decorators.method_decorator(login_required)
    def dispatch(self, *args, **kwargs):
        """Make sure only authenticated users can access this tab."""
        return super(HamTab, self).dispatch(*args, **kwargs)

    @property
    def tab_visible(self):
        """Show tab only if current user is logged in."""
        return self.current_tab.request.user.is_authenticated()


class HiddenTab(MainNavigationBaseTab):
```

```
    """A hidden TabView."""
    _is_tab = True
    tab_id = 'hidden'
    template_name = 'hidden_tab.html'
```

In this example, a base tab class was created. Because it does not contain the `_is_tab` class attribute, it is not listed as a tab itself (which wouldn't be possible anyway, as it has no `tab_id`). The three classes `SpamTab`, `HamTab` and `HiddenTab` extend the `MainNavigationBaseTab`. The base class predefines a tab group, so each extending tab doesn't have to define it again, therefore following the DRY principle. It also adds a new context variable called `spam` to the context of each tab.

The second tab, `HamTab`, overrides some more attributes. In this example, the tab is only visible in the template if the current user is logged in. Additionally, if the user is logged in, a new CSS class *logged_in_only* gets added to the `tab_classes` list, in order to be able to show the user that this is a "secret" tab that guest users aren't able to see. A copy of the `tab_classes` list is used because otherwise the CSS class would be added to all classes which extend `MainNavigationBaseTab`.

The third tab, `HiddenTab`, doesn't define a `tab_label` and is therefore not shown at all (see default behavior of `views.TabView.tab_visible()`).

Warning: Keep in mind that if you're overriding `get_context_data(self, **kwargs)`, you need to call the superclasses' versions of the method first (like in the example above). Otherwise, you'll override the `tabs` context variable.

Accessing request data

If you want to access `self.request` in a function used to render the tab item in your template, you may notice that it is not available. This is because the tab instances other than your current tab don't pass through the request dispatching functions.

If you need access to your current request information, you can access it via the `self.current_tab` attribute, e.g.:

```
class SpamTab(TabView):
    # (...)
    def username(self):
        current_tab = self.current_tab
        user = current_tab.request.user
        return user.username
```

1.2.2 Tab navigation template

Available context variables:

- `tabs`
- `current_tab_id`
- `parent_tabs`
- `parent_tab_id`
- `child_tabs`
- `view`

In order to display the tabs in your templates, you need to create a tab list using the `{{ tabs }}` context variable. You can also use `{{ current_tab_id }}` to access the id of the currently active tab. Here is an example template:

```
<div id="tab_navigation">
    <ul>
        {% for tab in tabs %}
            <li class="{{ tab.tab_classes|join:" " }}{% if tab.tab_id == current_tab_id %} active
                <a href="/tabs/{{ tab.tab_id }}/" {%if tab.tab_rel %}rel="{{ tab.tab_rel }}"{% en
                {% if tab.tab_counter %}<em>{{ tab.tab_counter }}</em>{% endif %}
                {{ tab.tab_label }}
                </a>
            </li>
        {% endfor %}
    </ul>
</div>
```

Each item in the `{{ tabs }}` list is an instance of a tab in the same tab group as the current tab. Therefore you can use all class- and instance variables as well as all functions without arguments that are defined in the `views.TabView` base class or in the extending class.

If you want to access the current tab instance, you can simply use the `view` variable which is provided by Django's ContextMixin.

It's a good idea to put this template code in a file called e.g. `blocks/tabination.html` and to include it everywhere you want the navigation to be displayed:

```
...
{% include "blocks/tabination.html" %}
...
```

1.2.3 Multilevel navigation

django-tabination can also be used for multilevel navigation. You can use the `tab_parent` attribute to connect two navigation levels. The attribute is defined at the **child base navigation class**. The following example has a tab called `ParentTab` which is at the first navigation level. The base class of the second navigation level is `ChildNavigationBaseTab`. This class defines the attribute `tab_parent` to connect itself and all it's siblings with the parent navigation level.

```
from tabination.views import TabView


# First navigation level

class ParentNavigationBaseTab(TabView):
    """Base class for all parent navigation tabs."""
    tab_group = 'parent_navigation'
    tab_classes = ['parent-navigation-tab']


class ParentTab(ParentNavigationBaseTab):
    _is_tab = True
    tab_id = 'parent'
    tab_label = 'Parent'
    template_name = 'parent_tab.html'


class EmptyTab(ParentNavigationBaseTab):
    _is_tab = True
    tab_id = 'empty'
    tab_label = 'Empty'
    template_name = 'empty_tab.html'


# Second navigation level
```

```
class ChildNavigationBaseTab(TabView):
    """Base class for all child navigation tabs."""
    tab_group = 'child_navigation'
    tab_classes = ['child-navigation-tab']
    tab_parent = ParentTab


class FirstChildTab(ChildNavigationBaseTab):
    _is_tab = True
    tab_id = 'first_child'
    tab_label = 'First Child'
    template_name = 'first_child_tab.html'


class SecondChildTab(ChildNavigationBaseTab):
    _is_tab = True
    tab_id = 'second_child'
    tab_label = 'Second Child'
    template_name = 'second_child_tab.html'
```

Multilevel template context

If you use multilevel navigation new values are added to your template context.

If the current tab has a parent tab the following values are added:

parent_tab_id The `tab_id` of the parent tab.

parent_tabs Instances of all tabs at the parent level.

The following variable is added to the template context if the current tab is a parent tab and has one or more children:

child_tabs A list of instances of all child tabs.

Because the `{{ current_tab }}` and `{{ current_tab_id }}` context variables always refer to the globally current tab and not to the active tab in the current tab group, you would have to write different templates for the different levels of navigation to properly set an `active` class on the tab item. To avoid this problem, you can use the `tab.group_current_tab` attribute which is provided with every tab object and refers to the active tab of the current tab group, no whether where in the hierarchy the group is positioned.

If you didn't quite understand the things above (it's complicated I know...), just take a look at the following example:

```
{# blocks/tab.html #}

<li class="{{ tab.tab_classes|join:" " }}{% if tab.tab_id == tab.group_current_tab.tab_id %} acti
    <a href="/{{ tab.tab_id }}/" {%if tab.tab_rel %}rel="{{ tab.tab_rel }}"{% endif %}>
    {{ tab.tab_label }}
    </a>
</li>
```

```
{# blocks/navigation.html #}

<div id="tab_navigation">
    {% if parent_tabs %}
    <ul>
            {% for tab in parent_tabs %}
                {% include 'blocks/tab.html' %}
            {% endfor %}
    </ul>
    {% endif %}
    <ul>
```

```
        {% for tab in tabs %}
            {% include 'blocks/tab.html' %}
        {% endfor %}
    </ul>
    {% if child_tabs %}
    <ul>
        {% for tab in child_tabs %}
            {% include 'blocks/tab.html' %}
        {% endfor %}
    </ul>
    {% endif %}
</div>
```

1.2.4 Sorting tabs

Tabs are sorted by their `weight` attribute automatically. Tabs with a lower weight are sorted before tabs with a higher weight. The default value of `weight` is 0. Negative values are also allowed and will be sorted before postive values. If two tabs have the same weight the natural order of the classes is used.

1.3 TabView

This page documents the attribute values and functions of the `TabView` base class.

1.4 Testing

Current build status: To set up a testing environment, you need to install Django and some additional dependencies:

```
$ pip install Django
$ make install
```

To run the test suite, use

```
$ make test
```

If you want to generate a coverage report, use

```
$ make report
```

To see a HTML version of the coverage report, there's

```
$ make report-html
```

Finally, to check conformance to the PEP8 coding standard, use

```
$ make flake8
```

Note: The flake8 configuration ignores E128 (*continuation line under-indented for visual indent*) errors and allows a max line length of 99 characters per line.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search