django-adminlinks Documentation

Release 0.8.2

Keryn Knight

February 13, 2014

Contents

CHAPTER 1

Usage documentation

1.1 Getting started with django-adminlinks

For the purposes of this document, it is assumed that things like pip and virtualenv are being used, as doing Django projects without them is almost unthinkable.

1.1.1 Prerequisite packages

Beyond those recommendations above, django-adminlinks has the current dependencies.

- django-classy-tags >= 0.3.4.1
- Django >= 1.4

The Django version requirement

as far as I know, the only reason Django 1.2 & 1.3 don't work is because the built-in template tags for rendering the *CSS* and *JS* both make use of `{% static %}`, where previously they'd have used `{{ STATIC_URL }}` or `{{ MEDIA_URL }}`

1.1.2 Installation

Because I don't believe in polluting the PyPI with half-finished or abandonware installables, the only way to install django-adminlinks at this time is either via cloning the Git repository directly into your pythonpath, or having pip do it for you:

```
pip install git+https://github.com/kezabelle/django-adminlinks.git@0.8.0
```

Once the package is installed, you'll need to update your Django project settings (usually `settings.py`) and add `adminlinks` to your `INSTALLED_APPS`, so that the template tags are available:

```
INSTALLED_APPS = (
    # These are all required.
    'django.contrib.auth',
    'django.contrib.admin',
    'django.contrib.contenttypes',
    [...]
    # our new app!
    'adminlinks',
```

```
    [...]
)
```

1.1.3 Basic Usage

Wherever you want to link to an object's admin from a template, you'll need to have loaded the correct template tags. Mostly, that means throwing the following in at the top of the template you want to display links in:

```
{% load adminlinks_buttons %}
```

With the buttons loaded for the template, you can sprinkle whatever links you want for any valid object; in the following example, the context variable *object* is a model instance:

```
<div class="headline">{{ object.title }}</div>

<!-- Your basic actions -->
{% render_edit_button object %}
{% render_delete_button object %}

<!-- Are there other tags available? Why yes, there are! -->
{% render_add_button object %}
{% render_history_button object %}
{% render_edit_field_button object 'title' %}

<!-- there's also a grouped button, which handles add/edit/delete/history -->
{% render_admin_buttons object %}

<!-- there's a button for going to the admin index, too -->
{% render_admin_button %}
{% render_admin_button "my_custom_admin" %}
```

Note: When we refer to a **valid object**, we generally mean a Django model or model instance.

1.2 Optional, replaceable extras

In the *installation and basic usage* we covered the bare minimum required to display plain text links to the admin. We can, of course, do better than that, and django-adminlinks comes with a few bits and pieces to do so. None of them are required, or even enabled by default, and all of them are replaceable.

1.2.1 Styling the links

If you're lucky, they'll already look as nice as your regular links, because that's all they are. Don't believe me? Here's the default template for the edit link:

```
{% if link %}
{% load i18n %}
<a href="{{ link }}" class="django-adminlinks--btn django-adminlinks--edit" data-adminlinks="autoclos
    {% blocktrans with verbose_name as name %}Edit this <span class="django-adminlinks--btn--importa
</a>
{% endif %}
```

As you can see, the links can be styled in a composite way because they have multiple classes.

Using the provided styles

There's a template tag we've not mentioned until now, because it's probably not suitable for use with things like django-sekizai or django-compressor.

Behold `render_adminlinks_css`:

```
{% load adminlinks_assets %}
<!doctype html>
<html>
    <head>
        {% render_adminlinks_css %}
    </head>
    <body>
        [...]
    </body>
</html>
```

That's all there is to it. Infact, it's not even a complex tag, it just handles rendering this on your behalf and does so if the `request.user` can potentially use the *AdminSite*:

```
{% if should_load_assets %}
    {% load static %}
    <link rel="stylesheet" media="screen" href="{% static 'adminlinks/css/widgets.css' %}{% if debug
    <link rel="stylesheet" media="screen" href="{% static 'adminlinks/css/fancyiframe-custom.css' %}
{% endif %}
```

The styles in `widgets.css` are designed to emulate the visuals of the default Django admin, without using any images. You can override them by providing your own `widgets.css`, or override the `adminlinks/templates/adminlinks/css.html` file in your own templates, wherever you've specified them under `TEMPLATE_DIRS`.

As you can see, the `{% render_adminlinks_css %}` also renders another stylesheet, `fancyiframe-custom.css`, which is used in conjunction with the *provided JavaScript*.

Including the CSS should get you links like this GIF, which is just the output of:

```
{% load adminlinks_buttons %}
{% render_admin_buttons blogpost %}
```

Note: If you're using either django-sekizai or django-compressor, you'll need to handle whether or not the stylesheets should be displayed yourself. The simplest test would be something like `{% if request.user.is_authenticated and request.user.is_staff %}`. Or you can just always include them, I suppose.

1.2.2 Making things modal

Though it is inevitably not perfect, we can make the UX a little better by allowing users to edit things *in-place*, via the included modal iframe.

Using the provided JavaScript

Exactly like the *bundled Stylesheets template tag*, there's a template tag for rendering the bundled JavaScript. The same caveats about django-sekizai and django-compressor apply:

```
{% load adminlinks_assets %}
<!doctype html>
<html>
    <head>
        [...]
    </head>
    <body>
    [...]
        {% render_adminlinks_js %}
    </body>
</html>
```

Which will output:

```
{% if should_load_assets %}
    {% load static %}
    <script type="text/javascript" src="{% static 'admin/js/jquery.min.js' %}{% if debug %}?cachebust
    <script type="text/javascript" src="{% static 'adminlinks/js/jquery.fancyiframe.js' %}{% if debu
    <script type="text/javascript" src="{% static 'adminlinks/js/adminlinks.js' %}{% if debug %}?cach
{% endif %}
```

As you can see, the JavaScript is a little bit more involved. It uses the jQuery which comes with Django, and a script of my own wrangling, to display an `<iframe>` in a modal box. It hooks up all classes of `django-adminlinks--btn` to this modal box – this CSS class is applied as a namespace to all the links Here's an example of it being used with the *default CSS* to do **per-field** editing of the *title*.

Note: The modal window has been sped up here to keep the animated GIF small, and the admin is in popup mode thanks to `fix_admin_popups()`.

1.2.3 Patching the standard ModelAdmin

If you're making use of the *bundled JavaScript*, through the template tag or otherwise, you'll probably want to alter the behaviour of the Django ModelAdmin instances in an effort to better support the modal-editing. We can extend the behaviour like so:

```
from django.contrib import admin
from adminlinks.admin import AdminlinksMixin
from myapp.models import MyModel

# At it's most simple, mixing in with the default modeladmin.
class MyModelAdmin(AdminlinksMixin, admin.ModelAdmin):
    list_display = ['my_field', 'my_other_field']
admin.site.register(MyModel, MyModelAdmin)
```

Or, for the slightly more complex usage of replacing a third-party admin:

```
from django.contrib import admin
from adminlinks.admin import AdminlinksMixin
from django.contrib.auth.models import User
from django.contrib.auth.admin import UserAdmin

# Replacing an existing Modeladmin
try:
    admin.site.unregister(User)
```

```
except admin.NotRegistered:
    pass

class MyUserAdmin(AdminlinksMixin, UserAdmin):
    pass
admin.site.register(User, MyUserAdmin)
```

For more complex admins, or different ways of handling displaying things (such as using jQuery ajaxForm, or one of the many other modal boxes) you'll have to go off the beaten track and drop some/most of the provided stuff. Any suggestions for how to make it more flexible, do *get in contact* and explain.

Editing field subsets

If your intention is to use the **edit field** template tag:

```
{% render_edit_field_button object 'title' %}
```

You'll need to amend your `ModelAdmin` to support the dynamic generation of that form, using the `AdminlinksMixin`, which updates the standard `get_urls()` to expose the `change_field_view()`

Success responses

To allow our *bundled JavaScript's modal box* to automatically close after a non-idempotent action (eg: add/change/delete), we need to override the existing modeladmin methods `response_add()`, `response_change()` and `delete_view()` to handle sending a message to the modal window. That too is covered by including `AdminlinksMixin`.

1.2.4 Simplifying the `AdminSite` visual clutter

If you're aiming for doing *everything* via the front-end, using the template tags to their fullest potential, you may want to get rid of some of the visual noise the admin provides (header, breadcrumbs, *etc*). Add the following to your `TEMPLATE_CONTEXT_PROCESSORS` to make it behave as if it were in a popup, reducing the visual context appropriately:

```
TEMPLATE_CONTEXT_PROCESSORS = (

    # These are other context processors we probably already have ...
    "django.contrib.auth.context_processors.auth",
    "django.core.context_processors.media",
    "django.core.context_processors.static",
    "django.core.context_processors.request",

    # This is our new context processor!
    "adminlinks.context_processors.fix_admin_popups",
)
```

See `fix_admin_popups()` for more.

CHAPTER 2

What it is

A suite of template tags for rendering links to a Django `AdminSite` instance.

At it's most basic, given a `Model`, it will do the appropriate checks to ensure that the currently signed in user can perform the requested action via the admin, and displays a configurable template with a link to the right place.

CHAPTER 3

Why?

Because I wedge the Django admin into everything, whether it should fit or not. Not so much because I love the admin, but because it provides a well-understood CRUD application that can be bolted onto in a pinch.

CHAPTER 4

Features

Here's a brief run-down on what's in the box:

- Basic, sane permission checking
  - Calling the template tags without a RequestContext should not expose any markup.
  - Users must be signed in, and pass the permission checking for the specific administration view.
- Optional *CSS* and *JavaScript* to improve the functionality by providing "button" like links, and a modal window for opening links.
- Pretty reasonable documentation. Or at least that's the aim.
- An additional view on all instances which subclass our `AdminlinksMixin`, to edit a specific field on a model, which can be used for some fairly neat in-place editing of only distinct parts of some data.

CHAPTER 5

Show me a demo!

The main draw, at least for me, is the ability to get frontend-editing for any model registered with the `AdminSite`. Currently when using the *CSS*, *JS* and *Modeladmin mixin*, you can hope for behaviour shown below.

5.1 Links

Figure 5.1: Here's how the various links may appear if using the *included CSS*. The example above is using the `{% render_admin_buttons %}` tag. See *Basic Usage* for more examples, and `Combined` for the auto-generated API documentation for the tag.

5.2 Modal editing

Figure 5.2: Using the *included CSS* and *JavaScript* gets you an iframe-based modal window. In the above GIF, the *Adminlinks Modeladmin Mixin* is being used to expose **per-field** editing of the *title* and **auto-closing on success**.

Note: The modal window has been sped up here to keep the animated GIF small, and the admin is in popup mode thanks to `fix_admin_popups()`.

CHAPTER 6

Developer documentation (API, etc)

6.1 All of the template tags

Listed below are reference materials for all the template tags considered public and suitable for use at this time. None are required.

6.1.1 Including shipped static assets

May be used in a template by adding the following line before calling any of them:

```
{% load adminlinks_assets %}
```

Or to put just some of the available tags in your template namespace you can do:

```
{% load render_adminlinks_js from adminlinks_assets %}
{% load render_adminlinks_css from adminlinks_assets %}
```

Asset tags

6.1.2 Rendering links to an admin site

May be used in a template by adding the following line before calling any of them:

```
{% load adminlinks_buttons %}
```

Or to put just some of the available tags in your template namespace you can do:

```
{% load render_edit_button from adminlinks_buttons %}
{% load render_changelist_button from adminlinks_buttons %}
```

and so on.

Link tags

class `adminlinks.templatetags.adminlinks_buttons.`**Add**(*parser*, *tokens*)

An `InclusionTag` to render a link to the `add_view()` for a `Model` mounted onto a `ModelAdmin` on the `AdminSite`:

```
{% render_add_button my_class %}
{% render_add_button my_class "my_custom_admin" %}
{% render_add_button my_class "my_custom_admin" "a=1&b=2&a=3" %}
```

get_link_context(*context*, *obj*, *admin_site*, *querystring*)
Adds a *link* and *verbose_name* to the context, if `is_valid()`

Parameters

- **context** – Hopefully, a `RequestContext` otherwise `is_valid()` is unlikely to be `True`
- **obj** – the `Model` class to link to. Must have `Options` from which we can retrieve a `verbose_name`
- **admin_site** – name of the admin site to use; defaults to **"admin"**
- **querystring** – a querystring to include in the link output. Defaults to "_popup=1"

Returns the link values.

Return type dictionary.

class `adminlinks.templatetags.adminlinks_buttons.`**BaseAdminLink**
Class for mixing into other classes to provide `is_valid()`, allowing subclasses to test the incoming data and react accordingly:

```
class MyContextHandler(BaseAdminLink):
    def get_context(self, context, obj):
        assert self.is_valid(context, obj) == True
```

Also provides `base_options` suitable for using in classy tags.

base_options = (<Argument: obj>, <StringArgument: admin_site>, <Argument: querystring>)
Default options involved in `InclusionTag` subclasses. Stored as a tuple because manipulating Options lists is more difficult than we'd like; see https://github.com/ojii/django-classy-tags/issues/14

get_context(*context*, *obj*, **args*, ***kwargs*)
Entry point for all subsequent tags. Tests the context and bails early if possible.

is_valid(*context*, *obj*, **args*, ***kwargs*)
Performs some basic tests against the parameters passed to it to ensure that work should carry on afterwards.

Parameters

- **context** – a `Context` subclass, or dictionary-like object which fulfils certain criteria. Usually a `RequestContext`.
- **obj** – the `Model`, either as a class or an instance. Or, more specifically, anything which as a `Options` object stored under the *_meta* attribute.

Returns whether or not the context and object pair are valid.

Return type `True` or `False`

See also:

`context_passes_test()`

class `adminlinks.templatetags.adminlinks_buttons.`**ChangeList**(*parser*, *tokens*)
An `InclusionTag` to render a link to the `changelist_view()` (paginated objects) for a `ModelAdmin` instance:

```
{% render_changelist_button my_class %}
{% render_changelist_button my_class "my_custom_admin" %}
{% render_changelist_button my_class "my_custom_admin" "a=1&b=2&a=3" %}
```

get_link_context(*context*, *obj*, *admin_site*, *querystring*)
Adds a *link* and *verbose_name* to the context, if `is_valid()`

Parameters

- **context** – Hopefully, a `RequestContext` otherwise `is_valid()` is unlikely to be `True`
- **obj** – the `Model` class to link to. Must have `Options` from which we can retrieve a `verbose_name`
- **admin_site** – name of the admin site to use; defaults to **"admin"**
- **querystring** – a querystring to include in the link output. Defaults to "pop=1" unless Django > 1.6, when it changes to "_popup=1"

Returns the link values.

Return type dictionary.

class adminlinks.templatetags.adminlinks_buttons.**Combined**(*parser*, *tokens*)
This needs reworking, I think.

get_link_context(*context*, *obj*, *admin_site*)
Wraps all the other adminlink template tags into one.

Parameters

- **context** – Hopefully, a `RequestContext` otherwise `is_valid()` is unlikely to be `True`
- **obj** – the `Model` class to link to. Must have `Options` from which we can retrieve a `verbose_name`
- **admin_site** – name of the admin site to use; defaults to **"admin"**

Returns the link values.

Return type dictionary.

class adminlinks.templatetags.adminlinks_buttons.**Delete**(*parser*, *tokens*)
An `InclusionTag` to render a link to the delete confirmation form for an object:

```
{% render_delete_button my_obj %}
{% render_delete_button my_obj "my_custom_admin" %}
{% render_delete_button my_obj "my_custom_admin" "a=1&b=2&a=3" %}
```

get_link_context(*context*, *obj*, *admin_site*, *querystring*)
Adds a *link* and *verbose_name* to the context, if `is_valid()`

Parameters

- **context** – Hopefully, a `RequestContext` otherwise `is_valid()` is unlikely to be `True`
- **obj** – the `Model` instance to link to. Must have a primary key, and `Options` from which we can retrieve a `verbose_name`
- **admin_site** – name of the admin site to use; defaults to **"admin"**
- **querystring** – a querystring to include in the link output. Defaults to "_popup=1"

Returns the link values.

Return type dictionary.

class adminlinks.templatetags.adminlinks_buttons.**Edit**(*parser*, *tokens*)
An InclusionTag to render a link to the admin change form for an object:

```
{% render_edit_button my_obj %}
{% render_edit_button my_obj "my_custom_admin" %}
{% render_edit_button my_obj "my_custom_admin" "a=1&b=2&a=3" %}
```

get_link_context(*context*, *obj*, *admin_site*, *querystring*)
Adds a *link* and *verbose_name* to the context, if is_valid()

Parameters

- **context** – Hopefully, a RequestContext otherwise is_valid() is unlikely to be True
- **obj** – the Model instance to link to. Must have a primary key, and Options from which we can retrieve a verbose_name
- **admin_site** – name of the admin site to use; defaults to **"admin"**
- **querystring** – a querystring to include in the link output. Defaults to "_popup=1"

Returns the link values.

Return type dictionary.

class adminlinks.templatetags.adminlinks_buttons.**EditField**(*parser*, *tokens*)
An InclusionTag to render a link to a customised admin change form for an object, showing only the requested field:

```
{% render_edit_field_button my_obj "field_name" %}
{% render_edit_field_button my_obj "field_name" "my_custom_admin" %}
{% render_edit_field_button my_obj "field_name" "my_custom_admin" "a=1&b=2&a=3" %}
```

Note: Use of this class requires that the ModelAdmin includes AdminlinksMixin or otherwise creates a named url ending in *change_field*.

Changed in version 0.8.1: The default template, adminlinks/edit_field_link.html now expects to be able to use {% load static %} if the field being edited is either a BooleanField or a NullBooleanField, so that it can render an icon.

get_link_context(*context*, *obj*, *fieldname*, *admin_site*, *querystring*)
Adds a *link* and *verbose_name* to the context, if is_valid()

Parameters

- **context** – Hopefully, a RequestContext otherwise is_valid() is unlikely to be True
- **obj** – the Model instance to link to. Must have a primary key, and Options from which we can retrieve a verbose_name
- **fieldname** – the specific model field to render a link for.
- **admin_site** – name of the admin site to use; defaults to **"admin"**
- **querystring** – a querystring to include in the link output. Defaults to "_popup=1"

Returns the link values.

Return type dictionary.

class adminlinks.templatetags.adminlinks_buttons.**History**(*parser*, *tokens*)
An InclusionTag to render a link to the object's history_view() in a ModelAdmin instance:

```
{% render_history_button my_obj %}
{% render_history_button my_obj "my_custom_admin" %}
{% render_history_button my_obj "my_custom_admin" "a=1&b=2&a=3" %}
```

get_link_context(*context*, *obj*, *admin_site*, *querystring*)
Adds a *link* and *verbose_name* to the context, if is_valid()

Parameters

- **context** – Hopefully, a RequestContext otherwise is_valid() is unlikely to be True
- **obj** – the Model instance to link to. Must have a primary key, and Options from which we can retrieve a verbose_name
- **admin_site** – name of the admin site to use; defaults to **"admin"**
- **querystring** – a querystring to include in the link output. Defaults to "_popup=1"

Returns the link values.

Return type dictionary.

template = u'adminlinks/history_link.html'
what gets rendered by this tag.

6.2 Admin classes

These provide additional functionality which may be depended on by *django-adminlinks* or related packages.

class adminlinks.admin.**AdminUrlWrap**
A minor helper for mixing into ModelAdmin instances, exposing the url wrapping function used in the standard get_urls().

_get_wrap()
Returns some magical decorated view that applies admin_view() to a ModelAdmin view:

```
from django.contrib.admin import ModelAdmin

class MyObj(AdminUrlWrap, ModelAdmin):

    def do_something(self):
        wrapper = self._get_wrap()
        wrapped_view = wrapper(self.my_custom_view)
        return wrapped_view
```

Returns a decorated view.

class adminlinks.admin.**AdminlinksMixin**
Our mixin, which serves two purposes:

•Allows for per-field editing through the use of the use of change_field_view().

–Per-field editing requires the *change* permission for the object.

–Per-field editing is not exposed in the AdminSite at all.

  - Per-field editing may be enabled in the frontend by using the `EditField` template tag

- Allows for the following views to be automatically closed on success, using a customisable template (see `get_success_templates()` for how template discovery works.)
  - The add view (`add_view()`)
  - The edit view (`change_view()`)
  - The delete view (`delete_view()`)
  - The edit-field view (`change_field_view()`)

change_field_view(**args*, ***kwargs*)
Allows a user to view a form with only one field (named in the URL args) to edit. All others are ignored.

data_changed(*querydict*)
Can be passed things like request.GET, or just dictionaries, whatever. This is our magic querystring variable.

New in version 0.8.1.

delete_view(*request*, *object_id*, *extra_context=None*)
Overrides the Django default, to try and provide a better experience for frontend editing when deleting an object successfully.

Ridiculously, there's no response_delete method to patch, so instead we're just going to do a similar thing and hope for the best.

get_changelist(*request*, ***kwargs*)
If the changelist hasn't been customised, lets just replace it with our own, which should allow us to track data changes without erroring.

New in version 0.8.1.

get_response_add_context(*request*, *obj*)
Provides a context for the template discovered by `get_success_templates()`. Only used when we could reliably determine that the request was in our JavaScript modal window, allowing us to close it automatically.

For clarity's sake, it should always return the minimum values represented here.

Returns Data which may be given to a template. Must be JSON serializable, so that a template may pass it back to the browser's JavaScript engine.

Return type a dictionary.

get_response_change_context(*request*, *obj*)
Provides a context for the template discovered by `get_success_templates()`. Only used when we could reliably determine that the request was in our JavaScript modal window, allowing us to close it automatically.

For clarity's sake, it should always return the minimum values represented here.

Returns Data which may be given to a template. Must be JSON serializable, so that a template may pass it back to the browser's JavaScript engine.

Return type a dictionary.

get_response_delete_context(*request*, *obj_id*, *extra_context*)
Provides a context for the template discovered by `get_success_templates()`. Only used when we could reliably determine that the request was in our JavaScript modal window, allowing us to close it automatically.

For clarity's sake, it should always return the minimum values represented here.

Note: At the point this is called, the original object no longer exists, so we are stuck trusting the *obj_id* given as an argument.

Changed in version Introduced: `extra_context` parameter.

Returns Data which may be given to a template. Must be JSON serializable, so that a template may pass it back to the browser's JavaScript engine.

Return type a dictionary.

`get_success_templates`(*request*)

Forces the attempted loading of the following:

- a template for this model.
- a template for this app.
- a template for any parent model.
- a template for any parent app.
- a guaranteed to exist template (the base success file)

Parameters **request** – The WSGIRequest

Returns list of strings representing templates to look for.

`maybe_fix_redirection`(*request*, *response*, *obj=None*)

This is a middleware-ish thing for marking whether a redirect needs to say data changed ... it's pretty complex, so has lots of comments.

New in version 0.8.1.

`response_add`(*request*, *obj*, **args*, ***kwargs*)

Overrides the Django default, to try and provide a better experience for frontend editing when adding a new object.

`response_change`(*request*, *obj*, **args*, ***kwargs*)

Overrides the Django default, to try and provide a better experience for frontend editing when editing an existing object.

`should_autoclose`(*request*)

New in version 0.8.1.

Returns Whether or not `_autoclose` was in the request and whether **Save** was pressed, or whether **Save and add another/continue editing** was.

`wants_to_autoclose`(*request*)

New in version 0.8.1.

Returns Whether or not `_autoclose` was in the request

`wants_to_continue_editing`(*request*)

New in version 0.8.1.

Returns Whether **Save** was pressed, or whether **Save and add another/continue editing** was.

6.3 View mixins

These provide additional functionality which may be useful when using *django-adminlinks*

class adminlinks.views.**ModelContext**

When working with things like ListView, DetailView or anything else that acts on a single model type, it may be useful to be able to access that model in the template, specifically so that the Add template tag may be used even when there is no actual instance of model available, given the class:

```
class MyView(ModelContext, DetailView):
    model = MyModel
```

it would now be possible to render the add button, even if there is no object in the context:

```
{% load adminlinks_buttons %}
<!-- model is not an instance, but a class -->
{% render_add_button model %}
```

In the slightly contrived example above, if the object didn't exist, DetailView would throw a Http404 anyway, but for demonstration purposes it should illustrate the purpose of ModelContext

get_context_data(***kwargs*)

Puts the model into the template context.

6.4 Utility functions for template tags

adminlinks.templatetags.utils.**_add_custom_link_to_context**(*admin_site*, *request*, *opts*, *permname*, *viewname*, *url_params*, *query=None*)

Like _add_link_to_context(), but allows for using a specific named permission, and any named url on the modeladmin, with optional url parameters.

Uses _add_link_to_context() internally once it has established the permissions are OK.

Always returns a dictionary with two keys, whose values may be empty strings.

Parameters

- **admin_site** – the string name of an admin site; eg: *admin*
- **request** – the current WSGIRequest
- **opts** – the *_meta* Options object to get the *app_label* and *module_name* for the desired URL.
- **permname** – The permission name to find; eg: *add*, *change*, *delete*
- **viewname** – The name of the view to find; eg: *changelist*, *donkey*
- **url_params** – a list of items to be passed as *args* to the underlying use of reverse.
- **query** – querystring to append.

Returns a dictionary containing *link* and *verbose_name* keys, whose values are the reversed URL and the display name of the object. Both may be blank.

Return type dictionary

adminlinks.templatetags.utils.**_add_link_to_context**(*admin_site*, *request*, *opts*, *permname*, *url_params*, *query=None*)

Find out if a model is in our known list and at has least 1 permission. If it's in there, try and reverse the URL to return a dictionary for the final Inclusion Tag's context.

Always returns a dictionary with two keys, whose values may be empty strings.

Parameters

- **admin_site** – the string name of an admin site; eg: *admin*
- **request** – the current `WSGIRequest`
- **opts** – the *_meta* Options object to get the *app_label* and *module_name* for the desired URL.
- **permname** – The permission name to find; eg: *add*, *change*, *delete*
- **url_params** – a list of items to be passed as *args* to the underlying use of reverse.
- **query** – querystring to append.

Returns a dictionary containing *link* and *verbose_name* keys, whose values are the reversed URL and the display name of the object. Both may be blank.

Return type `dictionary`

`adminlinks.templatetags.utils.`**`_admin_link_shortcut`**(*urlname*, *params=None*, *query=None*)

Minor wrapper around `reverse()`, catching the `NoReverseMatch` that may be thrown, and instead returning an empty unicode string.

Parameters

- **urlname** – the view, or named URL to be reversed
- **params** – any parameters (as *args*, not *kwargs*) required to create the correct URL.

Returns the URL discovered, or an empty string

Return type unicode string

`adminlinks.templatetags.utils.`**`_changelist_popup_qs`**()

If we're not at 1.6, the changelist uses "pop" in the querystring.

Changed in version 0.8.1: Returns a tuple of the querystring and a boolean of whether or not

`adminlinks.templatetags.utils.`**`_get_admin_site`**(*admin_site*)

Given the name of an `AdminSite` instance, try to resolve that into an actual object

---

Note: This function is exposed in the public API as `get_admin_site()`, which uses memoization to cache discovered `AdminSite` objects.

Parameters **admin_site** – the string name of an `AdminSite` named and mounted on the project.

Returns an `AdminSite` instance matching that given in the *admin_site* parameter.

Return type `AdminSite` or `None`

`adminlinks.templatetags.utils.`**`_resort_modeladmins`**(*modeladmins*)

A pulled-up-and-out version of the sorting the standard Django `AdminSite` does on the index view.

Parameters **modeladmins** – dictionary of modeladmins

Returns the same modeladmins, with their ordering changed.

Return type `list`

`adminlinks.templatetags.utils.`**`context_passes_test`**(*context*)

Given a context, determine whether a `User` exists, and if they see anything.

Changed in version 0.8.1: if DEBUG is True, then better error messages are displayed to the user, as a reminder of what settings need to be in place. Previously it was dependent on having a LOGGING configuration that would show the messages.

Parameters **context** – a RequestContext. Accepts any Context like object, but it explicitly tests for a *request* key and *request.user*

Returns whether or not the given context should allow further processing.

Return type True or False

adminlinks.templatetags.utils.**get_admin_site**(**args*)

The public API implementation of _get_admin_site(), wrapped to use memoization.

adminlinks.templatetags.utils.**get_registered_modeladmins**(*request*, *admin_site*)

Taken from AdminSite, find all ModelAdmin classes attached to the given admin and compile a dictionary of Model types visible to the current User, limiting the methods available (add/edit/history/delete) as appropriate.

Always returns a dictionary, though it may be empty, and thus evaluate as Falsy.

Parameters

- **request** – the current request, for permissions checking etc.
- **admin_site** – a concrete AdminSite named and mounted on the project.

Returns visible ModelAdmin classes.

Return type dictionary

6.5 Context processors

adminlinks.context_processors.**fix_admin_popups**(*request*)

Should you desire it, you can force the entire admin to behave as if it were in a popup. This may be useful if you're exposing the entire thing as a frontend-edited site.

It forces all of the admin to believe that the request included *_popup=1* (or *pop=1* for the changelist in *Django_ < 1.6*) and thus hides the header, breadcrumbs etc.

It also keeps track of whether or not it was really requested via a popup, by populating the context with is_really_popup, and it also detects whether the view is supposed to respond by closing a modal window on success by putting will_autoclose into the context.

Changed in version 0.8.1: Previously the function adminlinks.context_processors.force_admin_popups() used this name.

Note: If there is no user, or the user is not authenticated, the context will never contain any of the documented keys.

---

adminlinks.context_processors.**force_admin_popups**(*request*)

Should you desire it, you can force the entire admin to behave as if it were in a popup. This may be useful if you're exposing the entire thing as a frontend-edited site.

It forces all of the admin to believe that the request included *_popup=1* (or *pop=1* for the changelist in *Django_ < 1.6*) and thus hides the header, breadcrumbs etc.

It also keeps track of whether or not it was really requested via a popup, by populating the context with is_really_popup, and it also detects whether the view is supposed to respond by closing a modal window on success by putting will_autoclose into the context.

New in version 0.8.1: Previously this was known as `adminlinks.context_processors.fix_admin_popups()`, even though it didn't really *fix* anything.

Note: If there is no user, or the user is not authenticated, the context will never contain any of the documented keys.

`adminlinks.context_processors.`**`patch_admin_context`**(*request*, *valid*, *invalid*)

If there is no user, or the user is not authenticated, the context will never contain `valid`.

If the `AdminSite` in use isn't the default `django.contrib.admin.site`, it will also fail (being unable to reverse the default admin), which is hopefully fine, because you should probably handle things yourself, you magical person.

New in version 0.8.1: Hoisted functionality required for `adminlinks.context_processors.force_admin_popups()` and `adminlinks.context_processors.fix_admin_popups()` into a separate function, which tests whether to apply the context.

Returns `valid` or `invalid` parameter.

Return type dictionary.

6.6 Internal settings

The following constants are available within *django-adminlinks*, and may be used by packages depending on it. The following are considered public and static.

None of these settings is overridable from user-land `django.conf.settings`.

`adminlinks.constants.`**`AUTOCLOSING`** **= u'_autoclose'**
querystring key for figuring out if we want to close a popup.

`adminlinks.constants.`**`DATA_CHANGED`** **= u'_data_changed'**
querystring key for tracking changes which might not otherwise be broadcast by a success template being rendered.

`adminlinks.constants.`**`MODELADMIN_REVERSE`** **= u'%(namespace)s:%(app)s_%(module)s_%(view)s'**
The format for getting any admin url, in any single-level URL namespace.

See also:

`get_registered_modeladmins()`

See also:

`_add_custom_link_to_context()`

`adminlinks.constants.`**`PERMISSION_ATTRIBUTE`** **= u'has_%s_permission'**
Generic string format for getting methods on a `ModelAdmin` which define permissions

See also:

`get_registered_modeladmins()`

6.7 Crafting a release

- Using git extras, run `git changelog CHANGELOG`
  - Tidy up the changelog output from now until previous tag.

  - Set the version string to be the current one ...
- `python setup.py clean`
- Test `python setup.py sdist`
- Test `python setup.py bdist_wheel`
- Make sure `python setup.py --long-description | rst2html.py --halt=3 > /dev/null` works ok.
- run bumpversion major/minor/patch
  - Check the tag and commit.
- Push to `origin` (GitHub)
- In future, create PyPI releases (see here)
  - `python setup.py sdist upload -r pypi`
  - `python setup.py bdist_wheel upload -r pypi`

CHAPTER 7

Contributing

Please do!

The project is hosted on GitHub in the kezabelle/django-adminlinks repository. The main branch is *master*.

Bug reports and feature requests can be filed on the repository's issue tracker.

If something can be discussed in 140 character chunks, there's also my Twitter account.

CHAPTER 8

Similar projects

In the course of writing this, I have become aware of other packages tackling the same sort of thing:

- Martin Mahner's django-frontendadmin
- Yaco Sistemas' django-inplaceedit
- Ryan Berg's django-jumptoadmin
- Maxime Haineault's django-editlive
- Interaction Consortium's django-adminboost

If you're aware of any others working in the same space, let me know and I'll add them here.

CHAPTER 9

License

`django-adminlinks 0.8.0` is available under the terms of the Simplified BSD License (alternatively known as the FreeBSD License, or the 2-clause License). See the `LICENSE` file in the source distribution for a complete copy, or in the this release on GitHub.

a