django-admin-tools Documentation

Release 0.8.1

David Jean Louis

Jul 20, 2017

Contents

This documentation covers the latest release of django-admin-tools, a collection of extensions and tools for the Django administration interface, django-admin-tools includes:

- a full featured and customizable dashboard (for the admin index page and the admin applications index pages),
- a customizable menu bar,
- tools to make admin theming easier.

To get up and running quickly, consult the *quick-start guide*, which describes all the necessary steps to install django-admin-tools and configure it for the default setup. For more detailed information about how to install and how to customize django-admin-tools, read through the documentation listed below.

Contents:

CHAPTER 1

Quick start guide

Before installing django-admin-tools, you'll need to have a copy of Django already installed. For the 0.8 release, Django 1.7 or newer is required.

Note: *Important note to users of django 1.6 or below:* starting from 0.6.0, django-admin-tools is *NOT* compatible with django <= 1.6. If you want, you can still use the 0.5.2 version that will always be available on Pypi.

Installing django-admin-tools

django-admin-tools requires Django version 1.3 or superior, optionally, if you want to display feed modules, you'll also need the Universal Feed Parser module.

There are several ways to install django-admin-tools, this is explained in *the installation section*.

For the impatient, the easiest method is to install django-admin-tools via easy_install or pip.

Using `easy_install`, type:

```
easy_install -Z django-admin-tools
```

Note that the `-Z` flag is required, to tell `easy_install` not to create a zipped package; zipped packages prevent certain features of Django from working properly.

Using `pip`, type:

```
pip install django-admin-tools
```

Basic configuration

For a more detailed guide on how to configure django-admin-tools, please consult *the configuration section*.

Prerequisite

In order to use django-admin-tools you obviously need to have configured your Django admin site. If you didn't, please refer to the relevant django documentation.

Configuration

First make sure you have the `django.core.context_processors.request` template context processor in your `TEMPLATE_CONTEXT_PROCESSORS` or `TEMPLATES` settings variable

Then add the `admin_tools.template_loaders.Loader` template loader to your `TEMPLATE_LOADERS` or `TEMPLATES` settings variable.

Note: Starting from django 1.8, `TEMPLATE_CONTEXT_PROCESSORS` and `TEMPLATE_LOADERS` are deprecated, they are replaced by the `TEMPLATES` variable, please refer to the relevant django documentation.

Note: Windows users: due to filename restrictions on windows platforms, you have to put the `admin_tools.template_loaders.Loader` at the very begining of the list in your `TEMPLATES` or `TEMPLATE_LOADERS` settings variable.

Then, add admin_tools and its modules to the `INSTALLED_APPS` like this:

```
INSTALLED_APPS = (
    'admin_tools',
    'admin_tools.theming',
    'admin_tools.menu',
    'admin_tools.dashboard',
    'django.contrib.auth',
    'django.contrib.sites',
    'django.contrib.admin'
    # ...other installed applications...
)
```

Important: it is very important that you put the admin_tools modules **before** the `django.contrib.admin` module, because django-admin-tools overrides the default Django admin templates, and this will not work otherwise.

Then, just add django-admin-tools to your urls.py file:

```
urlpatterns = patterns('',
    url(r'^admin_tools/', include('admin_tools.urls')),
    #...other url patterns...
)
```

Finally simply run:

```
python manage.py migrate
```

To collect static files run:

```
python manage.py collectstatic
```

Important: it is very important that `django.contrib.staticfiles.finders.AppDirectoriesFinder` be there in your `STATICFILES_FINDERS`.

Testing your new shiny admin interface

Congrats! At this point you should have a working installation of django-admin-tools. Now you can just login to your admin site and see what changed.

django-admin-tools is fully customizable, but this is out of the scope of this quickstart. To learn how to customize django-admin-tools modules please read *the customization section*.

CHAPTER 2

Installation guide

Requirements

Before installing django-admin-tools, you'll need to have a copy of Django already installed. For the 0.8 release, Django 1.7 or newer is required.

Note: *Important note to users of django 1.6 or below:* starting from 0.6.0, django-admin-tools is *NOT* compatible with django <= 1.6. If you want, you can still use the 0.5.2 version that will always be available on Pypi.

For further information, consult the Django download page, which offers convenient packaged downloads and installation instructions.

Note: If you want to display feeds in the admin dashboard, using the `FeedDashboardModule` you need to install the Universal Feed Parser module.

Installing django-admin-tools

There are several ways to install django-admin-tools:

- Automatically, via a package manager.
- Manually, by downloading a copy of the release package and installing it yourself.
- Manually, by performing a Mercurial checkout of the latest code.

It is also highly recommended that you learn to use virtualenv for development and deployment of Python software; `virtualenv` provides isolated Python environments into which collections of software (e.g., a copy of Django, and the necessary settings and applications for deploying a site) can be installed, without conflicting with other installed software. This makes installation, testing, management and deployment far simpler than traditional site-wide installation of Python packages.

Automatic installation via a package manager

Several automatic package-installation tools are available for Python; the most popular are easy_install and pip. Either can be used to install django-admin-tools.

Using `easy_install`, type:

```
easy_install -Z django-admin-tools
```

Note that the `-Z` flag is required, to tell `easy_install` not to create a zipped package; zipped packages prevent certain features of Django from working properly.

Using `pip`, type:

```
pip install django-admin-tools
```

It is also possible that your operating system distributor provides a packaged version of django-admin-tools. Consult your operating system's package list for details, but be aware that third-party distributions may be providing older versions of django-admin-tools, and so you should consult the documentation which comes with your operating system's package.

Manual installation from a downloaded package

If you prefer not to use an automated package installer, you can download a copy of django-admin-tools and install it manually. The latest release package can be downloaded from django-admin-tools's listing on the Python Package Index.

Once you've downloaded the package, unpack it (on most operating systems, simply double-click; alternately, type `tar zxvf django-admin-tools-X-Y-Z.tar.gz` at a command line on Linux, Mac OS X or other Unix-like systems). This will create the directory `django-admin-tools-X-Y-Z`, which contains the `setup.py` installation script. From a command line in that directory, type:

```
python setup.py install
```

Note: On some systems you may need to execute this with administrative privileges (e.g., `sudo python setup.py install`).

Manual installation from a git checkout

If you'd like to try out the latest in-development code, you can obtain it from the django-admin-tools repository, which is hosted on Github. To obtain the latest code and documentation, you'll need to have Git installed, at which point you can type:

```
git clone https://github.com/django-admin-tools/django-admin-tools.git
```

This will create a copy of the django-admin-tools Git repository on your computer; you can then add the `django-admin-tools` directory to your Python import path, or use the `setup.py` script to install as a package.

CHAPTER 3

Configuring django-admin-tools

Basic configuration

Once installed, you can add django-admin-tools to any Django-based project you're developing.

django-admin-tools is composed of several modules:

- admin_tools.theming: an app that makes it easy to customize the look and feel of the admin interface;
- admin_tools.menu: a customizable navigation menu that sits on top of every django administration index page;
- admin_tools.dashboard: a customizable dashboard that replaces the django administration index page.

Prerequisite

In order to use django-admin-tools you obviously need to have configured your django admin site, if you didn't, please refer to the relevant django documentation.

Required settings

First make sure you have the `django.core.context_processors.request` template context processor in your `TEMPLATE_CONTEXT_PROCESSORS` or `TEMPLATES` settings variable

Then add the `admin_tools.template_loaders.Loader` template loader to your `TEMPLATE_LOADERS` or `TEMPLATES` settings variable.

Note: Starting from django 1.8, `TEMPLATE_CONTEXT_PROCESSORS` and `TEMPLATE_LOADERS` are deprecated, they are replaced by the `TEMPLATES` variable, please refer to the relevant django documentation.

Note: Windows users: due to filename restrictions on windows platforms, you have to put the `admin_tools.template_loaders.Loader` at the very begining of the list in your `TEMPLATES` or `TEMPLATE_LOADERS` settings variable.

Then, add the django-admin-tools modules to the `INSTALLED_APPS` like this:

```
INSTALLED_APPS = (
    'admin_tools',
    'admin_tools.theming',
    'admin_tools.menu',
    'admin_tools.dashboard',
    'django.contrib.auth',
    'django.contrib.sites',
    'django.contrib.admin'
    # ...other installed applications...
)
```

Note: it is very important that you put the admin_tools modules **before** the `django.contrib.admin module`, because django-admin-tools overrides the default django admin templates, and this will not work otherwise.

django-admin-tools is modular, so if you want to disable a particular module, just remove or comment it in your `INSTALLED_APPS`. For example, if you just want to use the dashboard:

```
INSTALLED_APPS = (
    'admin_tools',
    'admin_tools.dashboard',
    'django.contrib.auth',
    'django.contrib.sites',
    'django.contrib.admin'
    # ...other installed applications...
)
```

Setting up the database

To set up the tables that django-admin-tools uses you'll need to type:

```
python manage.py migrate
```

Adding django-admin-tools to your urls.py file

You'll need to add django-admin-tools to your urls.py file:

```
urlpatterns = patterns('',
    url(r'^admin_tools/', include('admin_tools.urls')),
    #...other url patterns...
)
```

Collecting the Static Files

To collect static files run:

```
python manage.py collectstatic
```

Important: it is very important that `django.contrib.staticfiles.finders.AppDirectoriesFinder'' be there in your ``STATICFILES_FINDERS`.

Available settings variables

ADMIN_TOOLS_MENU The path to your custom menu class, for example "yourproject.menu.CustomMenu".

ADMIN_TOOLS_INDEX_DASHBOARD The path to your custom index dashboard, for example "yourproject.dashboard.CustomIndexDashboard".

ADMIN_TOOLS_APP_INDEX_DASHBOARD The path to your custom app index dashboard, for example "yourproject.dashboard.CustomAppIndexDashboard".

ADMIN_TOOLS_THEMING_CSS The path to your theming css stylesheet, relative to your STATIC_URL, for example:

```
ADMIN_TOOLS_THEMING_CSS = 'css/theming.css'
```

CHAPTER 4

Customization of the django-admin-tools modules

Introduction

django-admin-tools is very easy to customize, you can override the admin menu, the index dashboard and the app index dashboard.

For this django-admin-tools provides two management commands:

- `custommenu`
- `customdashboard`

Customizing the navigation menu

To customize the admin menu, the first step is to do the following:

```
python manage.py custommenu
```

This will create a file named `menu.py` in your project directory. If for some reason you want another file name, you can do:

```
python manage.py custommenu somefile.py
```

The created file contains a class that is a copy of the default menu, it is named `CustomMenu`, you can rename it if you want but if you do so, make sure you put the correct class name in your ADMIN_TOOLS_MENU settings variable.

Note: You could have done the above by hand, without using the `custommenu` management command, but it's simpler with it.

Now you need to tell django-admin-tools to use your custom menu instead of the default one, open your settings.py file and add the following:

```
ADMIN_TOOLS_MENU = 'yourproject.menu.CustomMenu'
```

Obviously, you need to change "yourproject" to the real project name, if you have chosen a different file name or if you renamed the menu class, you'll also need to change the above string to reflect your modifications.

At this point the menu displayed in the admin is your custom menu, now you can read *the menu and menu items API documentation* to learn how to create your custom menu.

Customizing the dashboards

To customize the index and app index dashboards, the first step is to do the following:

```
python manage.py customdashboard
```

This will create a file named `dashboard.py` in your project directory. If for some reason you want another file name, you can do:

```
python manage.py customdashboard somefile.py
```

The created file contains two classes:

- The `CustomIndexDashboard` class that corresponds to the admin index page dashboard;
- The `CustomAppIndexDashboard` class that corresponds to the index page of each installed application.

You can rename theses classes if you want but if you do so, make sure adjust the `ADMIN_TOOLS_INDEX_DASHBOARD` and `ADMIN_TOOLS_APP_INDEX_DASHBOARD` settings variables to match your class names.

Note: You could have done the above by hand, without using the `customdashboard` management command, but it's simpler with it.

Now you need to tell django-admin-tools to use your custom dashboard(s). Open your settings.py file and add the following:

```
ADMIN_TOOLS_INDEX_DASHBOARD = 'yourproject.dashboard.CustomIndexDashboard'
ADMIN_TOOLS_APP_INDEX_DASHBOARD = 'yourproject.dashboard.CustomAppIndexDashboard'
```

If you only want a custom index dashboard, you would just need the first line. Obviously, you need to change "yourproject" to the real project name, if you have chosen a different file name or if you renamed the dashboard classes, you'll also need to change the above string to reflect your modifications.

At this point the dashboards displayed in the index and the app index should be your custom dashboards, now you can read *the dashboard and dashboard modules API documentation* to learn how to create your custom dashboard.

Customizing the theme

Warning: The theming support is still very basic, do not rely too much on it for the moment.

This is very simple, just configure the `ADMIN_TOOLS_THEMING_CSS` to point to your custom css file, for example:

```
ADMIN_TOOLS_THEMING_CSS = 'css/theming.css'
```

A good start is to copy the `admin_tools/media/admin_tools/css/theming.css` to your custom file and to modify it to suits your needs.

CHAPTER 5

Working with multiple admin sites

Introduction

Django supports custom admin sites, and of course you can have as many admin sites as you want, django-admin-tools provides basic support for this, you can setup a custom dashboard or menu for each admin site.

Setting up a different dashboard and menu for each admin site instance

In the following example we will assume that you have two admin site instances: the default django admin site and a custom admin site of your own. In your urls, you should have something like this:

```
from django.conf.urls.defaults import *
from django.contrib import admin
from yourproject.admin import admin_site

admin.autodiscover()

urlpatterns = patterns('',
    (r'^admin/', include(admin.site.urls)),
    (r'^myadmin/', include(admin_site.urls)),
)
```

Now to configure your dashboards, you could do:

```
python manage.py customdashboard django_admin_dashboard.py
python manage.py customdashboard my_admin_dashboard.py
```

And to tell django-admin-tools to use your custom dashboards depending on the admin site being used, you just have to add the following to your project settings file:

```
ADMIN_TOOLS_INDEX_DASHBOARD = {
    'django.contrib.admin.site': 'yourproject.django_admin_dashboard.
↪CustomIndexDashboard',
    'yourproject.admin.admin_site': 'yourproject.my_admin_dashboard.
↪CustomIndexDashboard',
}
```

Note that the same applies for the `ADMIN_TOOLS_APP_INDEX_DASHBOARD` settings variable.

Finally do the same thing for menu:

```
python manage.py custommenu django_admin_menu.py
python manage.py custommenu my_admin_menu.py
```

And to tell django-admin-tools to use your custom menu depending on the admin site being used:

```
ADMIN_TOOLS_MENU = {
    'django.contrib.admin.site': 'yourproject.django_admin_menu.CustomMenu',
    'yourproject.admin.admin_site': 'yourproject.my_admin_menu.CustomMenu',
}
```

CHAPTER 6

The django-admin-tools menu and menu items API

This section describe the API of the django-admin-tools menu and menu items. Make sure you read this before creating your custom menu.

The `Menu` class

class `admin_tools.menu.`**`Menu`**(***kwargs*)

This is the base class for creating custom navigation menus. A menu can have the following properties:

`template` A string representing the path to template to use to render the menu. As for any other template, the path must be relative to one of the directories of your `TEMPLATE_DIRS` setting. Default value: "admin_tools/menu/menu.html".

`children` A list of children menu items. All children items mus be instances of the *`MenuItem`* class.

If you want to customize the look of your menu and it's menu items, you can declare css stylesheets and/or javascript files to include when rendering the menu, for example:

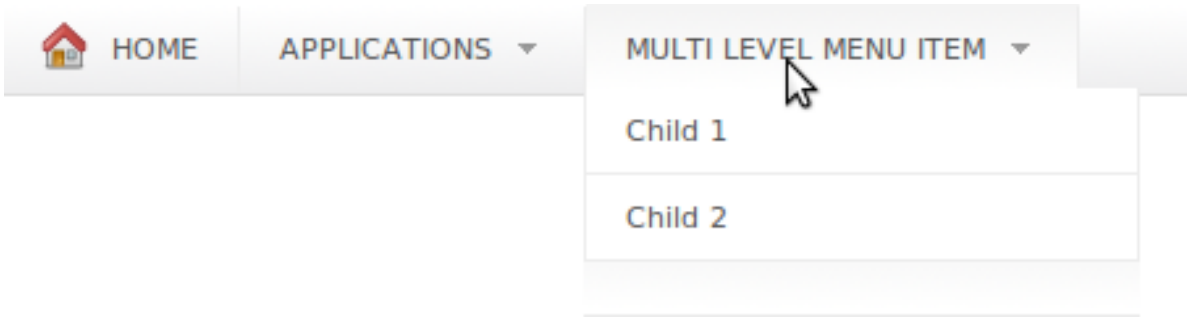
```
from admin_tools.menu import Menu

class MyMenu(Menu):
    class Media:
        css = {'all': ('css/mymenu.css',)}
        js = ('js/mymenu.js',)
```

Here's a concrete example of a custom menu:

```
from django.core.urlresolvers import reverse
from admin_tools.menu import items, Menu

class MyMenu(Menu):
    def __init__(self, **kwargs):
        super(MyMenu, self).__init__(**kwargs)
        self.children += [
```

```
            items.MenuItem('Home', reverse('admin:index')),
            items.AppList('Applications'),
            items.MenuItem('Multi level menu item',
                children=[
                    items.MenuItem('Child 1', '/foo/'),
                    items.MenuItem('Child 2', '/bar/'),
                ]
            ),
        ]
```

Below is a screenshot of the resulting menu:



init_with_context(*context*)
Sometimes you may need to access context or request variables to build your menu, this is what the `init_with_context()` method is for. This method is called just before the display with a `django.template.RequestContext` as unique argument, so you can access to all context variables and to the `django.http.HttpRequest`.

The `MenuItem` class

class `admin_tools.menu.items.`**MenuItem**(*title=None*, *url=None*, ***kwargs*)
This is the base class for custom menu items. A menu item can have the following properties:

title String that contains the menu item title, make sure you use the django gettext functions if your application is multilingual. Default value: 'Untitled menu item'.

url String that contains the menu item URL. Default value: '#'.

css_classes A list of css classes to be added to the menu item `li` class attribute. Default value: [].

accesskey The menu item accesskey. Default value: None.

description An optional string that will be used as the `title` attribute of the menu-item `a` tag. Default value: None.

enabled Boolean that determines whether the menu item is enabled or not. Disabled items are displayed but are not clickable. Default value: True.

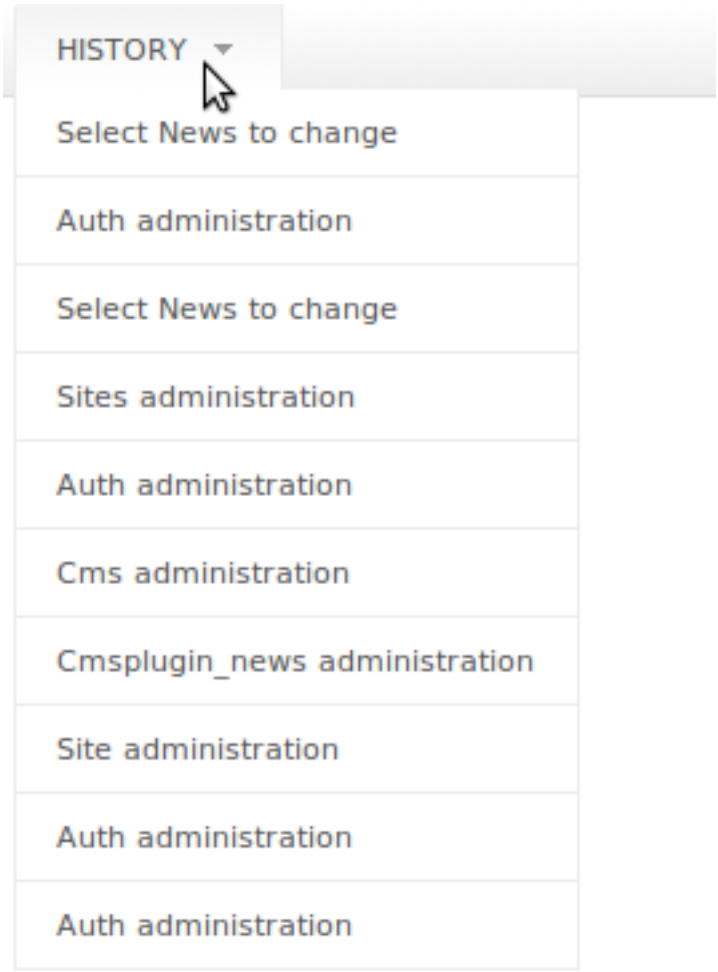
template The template to use to render the menu item. Default value: 'admin_tools/menu/item.html'.

children A list of children menu items. All children items must be instances of the `MenuItem` class.

init_with_context(*context*)
Like for menus, menu items have a `init_with_context` method that is called with a `django.template.RequestContext` instance as unique argument. This gives you enough flexibility to build complex items, for example, let's build a "history" menu item, that will list the last ten visited pages:

```
from admin_tools.menu.items import MenuItem

class HistoryMenuItem(MenuItem):
    title = 'History'

    def init_with_context(self, context):
        request = context['request']
        # we use sessions to store the visited pages stack
        history = request.session.get('history', [])
        for item in history:
            self.children.append(MenuItem(
                title=item['title'],
                url=item['url']
            ))
        # add the current page to the history
        history.insert(0, {
            'title': context['title'],
            'url': request.META['PATH_INFO']
        })
        if len(history) > 10:
            history = history[:10]
        request.session['history'] = history
```

Here's a screenshot of our history item:



is_empty()
Helper method that returns `True` if the menu item is empty. This method always returns `False` for basic items, but can return `True` if the item is an AppList.

is_selected(*request*)
Helper method that returns `True` if the menu item is active. A menu item is considered as active if it's URL or one of its descendants URL is equals to the current URL.

The `AppList` class

class `admin_tools.menu.items.`**AppList**(*title=None*, ***kwargs*)
A menu item that lists installed apps an their models. In addition to the parent *`MenuItem`* properties, the `AppList` has two extra properties:

models A list of models to include, only models whose name (e.g. "blog.comments.Comment") match one of the strings (e.g. "blog.*") in the models list will appear in the menu item.

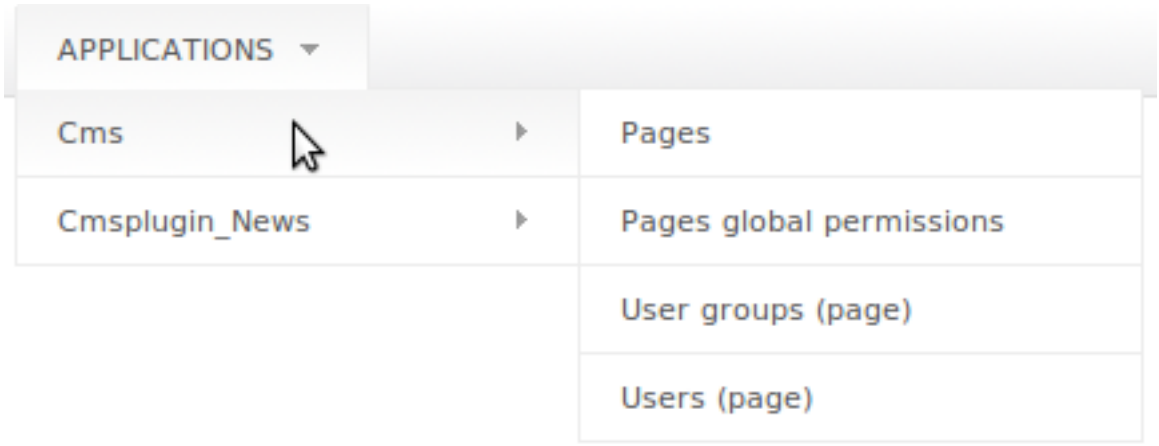
exclude A list of models to exclude, if a model name (e.g. "blog.comments.Comment") match an element of this list (e.g. "blog.comments.*") it won't appear in the menu item.

If no models/exclude list is provided, **all apps** are shown.

Here's a small example of building an app list menu item:

```
from admin_tools.menu import items, Menu

class MyMenu(Menu):
    def __init__(self, **kwargs):
        super(MyMenu, self).__init__(**kwargs)
        self.children.append(items.AppList(
            title='Applications',
            exclude_list=('django.contrib',)
        )
```

The screenshot of what this code produces:



Note: Note that this menu takes into account user permissions, as a consequence, if a user has no rights to change or add a `Group` for example, the `django.contrib.auth.Group model` child item won't be displayed in the menu.

init_with_context(*context*)
Please refer to *init_with_context()* documentation from *MenuItem* class.

is_empty()
Helper method that returns `True` if the applist menu item has no children.

```
>>> from admin_tools.menu.items import MenuItem, AppList
>>> item = AppList(title='My menu item')
>>> item.is_empty()
True
>>> item.children.append(MenuItem(title='foo'))
>>> item.is_empty()
False
>>> item.children = []
>>> item.is_empty()
True
```

The `ModelList` class

class `admin_tools.menu.items.`**`ModelList`**(*title=None*, *models=None*, *exclude=None*, ***kwargs*)

A menu item that lists a set of models. In addition to the parent *`MenuItem`* properties, the `ModelList` has two extra properties:

`models` A list of models to include, only models whose name (e.g. "blog.comments.Comment") match one of the strings (e.g. "blog.*") in the include list will appear in the dashboard module.

`exclude` A list of models to exclude, if a model name (e.g. "blog.comments.Comment" match an element of this list (e.g. "blog.comments.*") it won't appear in the dashboard module.

Here's a small example of building a model list menu item:

```
from admin_tools.menu import items, Menu

class MyMenu(Menu):
    def __init__(self, **kwargs):
        super(MyMenu, self).__init__(**kwargs)
        self.children += [
            items.ModelList(
                'Authentication', ['django.contrib.auth.*',]
            )
        ]
```

Note: Note that this menu takes into account user permissions, as a consequence, if a user has no rights to change or add a `Group` for example, the `django.contrib.auth.Group model` item won't be displayed in the menu.

`init_with_context`(*context*)

Please refer to *`init_with_context()`* documentation from *`MenuItem`* class.

`is_empty`()

Helper method that returns `True` if the modellist menu item has no children.

```
>>> from admin_tools.menu.items import MenuItem, ModelList
>>> item = ModelList(title='My menu item')
>>> item.is_empty()
True
>>> item.children.append(MenuItem(title='foo'))
>>> item.is_empty()
False
>>> item.children = []
>>> item.is_empty()
True
```

The `Bookmarks` class

class `admin_tools.menu.items.`**`Bookmarks`**(*title=None*, ***kwargs*)

A menu item that lists pages bookmarked by the user. This menu item also adds an extra button to the menu that allows the user to bookmark or un-bookmark the current page.

Here's a small example of adding a bookmark menu item:

```
from admin_tools.menu import items, Menu

class MyMenu(Menu):
    def __init__(self, **kwargs):
        super(MyMenu, self).__init__(**kwargs)
        self.children.append(items.Bookmarks('My bookmarks'))
```

init_with_context(*context*)
Please refer to *init_with_context()* documentation from *MenuItem* class.

is_selected(*request*)
A bookmark menu item is never considered as active, the real item is.

CHAPTER 7

The django-admin-tools dashboard and dashboard modules API

This section describe the API of the django-admin-tools dashboard and dashboard modules. Make sure you read this before creating your custom dashboard and custom modules.

..note:: If your layout seems to be broken or you have problems with included javascript files, you should try to reset your dashboard preferences (assuming a MySQL backend, the truncate command also works in postgress):

```
python manage.py dbshell
mysql> truncate admin_tools_dashboard_preferences;
```

For more information see this issue.

The `Dashboard` class

class `admin_tools.dashboard.`**`Dashboard`**(***kwargs*)

Base class for dashboards. The Dashboard class is a simple python list that has three additional properties:

`title` The dashboard title, by default, it is displayed above the dashboard in a `h2` tag. Default value: 'Dashboard'.

`template` The template to use to render the dashboard. Default value: 'admin_tools/dashboard/dashboard.html'

`columns` An integer that represents the number of columns for the dashboard. Default value: 2.

If you want to customize the look of your dashboard and it's modules, you can declare css stylesheets and/or javascript files to include when rendering the dashboard (these files should be placed in your media path), for example:

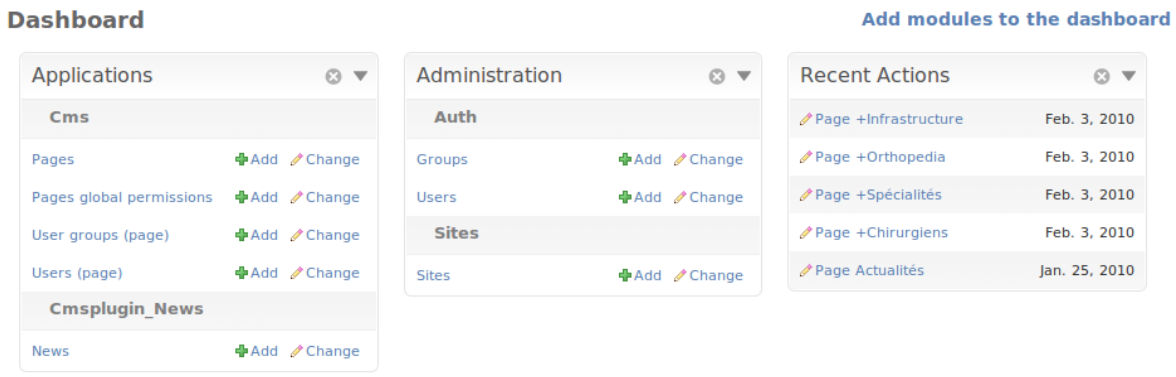
```
from admin_tools.dashboard import Dashboard

class MyDashboard(Dashboard):
    class Media:
        css = {
            'screen, projection': ('css/mydashboard.css',),
```

```
        }
        js = ('js/mydashboard.js',)
```

Here's an example of a custom dashboard:

```
from django.core.urlresolvers import reverse
from django.utils.translation import ugettext_lazy as _
from admin_tools.dashboard import modules, Dashboard

class MyDashboard(Dashboard):

    # we want a 3 columns layout
    columns = 3

    def __init__(self, **kwargs):

        # append an app list module for "Applications"
        self.children.append(modules.AppList(
            title=_('Applications'),
            exclude=('django.contrib.*',),
        ))

        # append an app list module for "Administration"
        self.children.append(modules.AppList(
            title=_('Administration'),
            models=('django.contrib.*',),
        ))

        # append a recent actions module
        self.children.append(modules.RecentActions(
            title=_('Recent Actions'),
            limit=5
        ))
```

Below is a screenshot of the resulting dashboard:



get_id()
: Internal method used to distinguish different dashboards in js code.

init_with_context(*context*)
: Sometimes you may need to access context or request variables to build your dashboard, this is what the `init_with_context()` method is for. This method is called just before the display with a `django.template.RequestContext` as unique argument, so you can access to all context variables and to

the `django.http.HttpRequest`.

The `AppIndexDashboard` class

class `admin_tools.dashboard.`**`AppIndexDashboard`**(*app_title*, *models*, ***kwargs*)

Class that represents an app index dashboard, app index dashboards are displayed in the applications index page. *`AppIndexDashboard`* is very similar to the *`Dashboard`* class except that its constructor receives two extra arguments:

`app_title` The title of the application

`models` A list of strings representing the available models for the current application, example:

```
['yourproject.app.Model1', 'yourproject.app.Model2']
```

It also provides two helper methods:

`get_app_model_classes()` Method that returns the list of model classes for the current app.

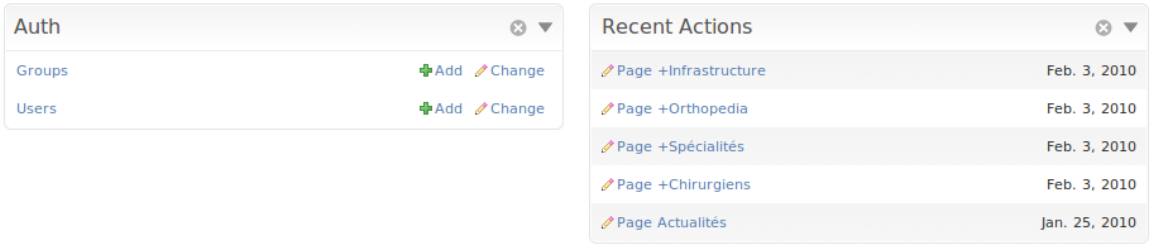
`get_app_content_types()` Method that returns the list of content types for the current app.

If you want to provide custom app index dashboard, be sure to inherit from this class instead of the *`Dashboard`* class.

Here's an example of a custom app index dashboard:

```
from django.core.urlresolvers import reverse
from django.utils.translation import ugettext_lazy as _
from admin_tools.dashboard import modules, AppIndexDashboard

class MyAppIndexDashboard(AppIndexDashboard):

    # we don't want a title, it's redundant
    title = ''

    def __init__(self, app_title, models, **kwargs):
        AppIndexDashboard.__init__(self, app_title, models, **kwargs)

        # append a model list module that lists all models
        # for the app and a recent actions module for the current app
        self.children += [
            modules.ModelList(self.app_title, self.models),
            modules.RecentActions(
                include_list=self.models,
                limit=5
            )
        ]
```

Below is a screenshot of the resulting dashboard:



get_app_content_types()
Return a list of all content_types for this app.

get_app_model_classes()
Helper method that returns a list of model classes for the current app.

get_id()
Internal method used to distinguish different dashboards in js code.

The `DashboardModule` class

class admin_tools.dashboard.modules.**DashboardModule**(*title=None*, ***kwargs*)
Base class for all dashboard modules. Dashboard modules have the following properties:

enabled Boolean that determines whether the module should be enabled in the dashboard by default or not. Default value: `True`.

draggable Boolean that determines whether the module can be draggable or not. Draggable modules can be re-arranged by users. Default value: `True`.

collapsible Boolean that determines whether the module is collapsible, this allows users to show/hide module content. Default: `True`.

deletable Boolean that determines whether the module can be removed from the dashboard by users or not. Default: `True`.

title String that contains the module title, make sure you use the django gettext functions if your application is multilingual. Default value: ‘’.

title_url String that contains the module title URL. If given the module title will be a link to this URL. Default value: `None`.

css_classes A list of css classes to be added to the module `div` class attribute. Default value: `None`.

pre_content Text or HTML content to display above the module content. Default value: `None`.

content The module text or HTML content. Default value: `None`.

post_content Text or HTML content to display under the module content. Default value: `None`.

template The template to use to render the module. Default value: ‘admin_tools/dashboard/module.html’.

init_with_context(*context*)
Like for the *Dashboard* class, dashboard modules have a `init_with_context` method that is called with a `django.template.RequestContext` instance as unique argument.

This gives you enough flexibility to build complex modules, for example, let’s build a “history” dashboard module, that will list the last ten visited pages:

```
from admin_tools.dashboard import modules

class HistoryDashboardModule(modules.LinkList):
    title = 'History'

    def init_with_context(self, context):
        request = context['request']
        # we use sessions to store the visited pages stack
        history = request.session.get('history', [])
        for item in history:
            self.children.append(item)
        # add the current page to the history
        history.insert(0, {
            'title': context['title'],
            'url': request.META['PATH_INFO']
        })
        if len(history) > 10:
            history = history[:10]
        request.session['history'] = history
```

Here's a screenshot of our history item:

is_empty()

Return True if the module has no content and False otherwise.

```
>>> mod = DashboardModule()
>>> mod.is_empty()
True
>>> mod.pre_content = 'foo'
>>> mod.is_empty()
False
>>> mod.pre_content = None
>>> mod.is_empty()
True
>>> mod.children.append('foo')
>>> mod.is_empty()
False
>>> mod.children = []
>>> mod.is_empty()
True
```

render_css_classes()
Return a string containing the css classes for the module.

```
>>> mod = DashboardModule(enabled=False, draggable=True,
...                       collapsible=True, deletable=True)
>>> mod.render_css_classes()
'dashboard-module disabled draggable collapsible deletable'
>>> mod.css_classes.append('foo')
>>> mod.render_css_classes()
'dashboard-module disabled draggable collapsible deletable foo'
>>> mod.enabled = True
>>> mod.render_css_classes()
'dashboard-module draggable collapsible deletable foo'
```

The Group class

class admin_tools.dashboard.modules.**Group**(*title=None*, ***kwargs*)
Represents a group of modules, the group can be displayed in tabs, accordion, or just stacked (default). As well as the *DashboardModule* properties, the *Group* has two extra properties:

display A string determining how the group should be rendered, this can be one of the following values: 'tabs' (default), 'accordion' or 'stacked'.

force_show_title Default behaviour for Group module is to force children to always show the title if Group has display = stacked. If this flag is set to False, children title is shown according to their''show_title'' property. Note that in this case is children responsibility to have meaningful content if no title is shown.

Here's an example of modules group:

```
from admin_tools.dashboard import modules, Dashboard

class MyDashboard(Dashboard):
    def __init__(self, **kwargs):
        Dashboard.__init__(self, **kwargs)
        self.children.append(modules.Group(
            title="My group",
            display="tabs",
            children=[
                modules.AppList(
                    title='Administration',
                    models=('django.contrib.*',)
                ),
                modules.AppList(
                    title='Applications',
                    exclude=('django.contrib.*',)
                )
            ]
        ))
```

The screenshot of what this code produces:

is_empty()
A group of modules is considered empty if it has no children or if all its children are empty.

```
>>> from admin_tools.dashboard.modules import DashboardModule, LinkList
>>> mod = Group()
>>> mod.is_empty()
True
>>> mod.children.append(DashboardModule())
>>> mod.is_empty()
True
>>> mod.children.append(LinkList('links', children=[
...     {'title': 'example1', 'url': 'http://example.com'},
...     {'title': 'example2', 'url': 'http://example.com'},
... ]))
>>> mod.is_empty()
False
```

The `LinkList` class

class admin_tools.dashboard.modules.**LinkList**(*title=None*, ***kwargs*)
A module that displays a list of links. As well as the *DashboardModule* properties, the *LinkList* takes an extra keyword argument:

layout The layout of the list, possible values are `stacked` and `inline`. The default value is `stacked`.

Link list modules children are simple python dictionaries that can have the following keys:

title The link title.

url The link URL.

external Boolean that indicates whether the link is an external one or not.

description A string describing the link, it will be the `title` attribute of the html `a` tag.

attrs Hash comprising attributes of the html `a` tag.

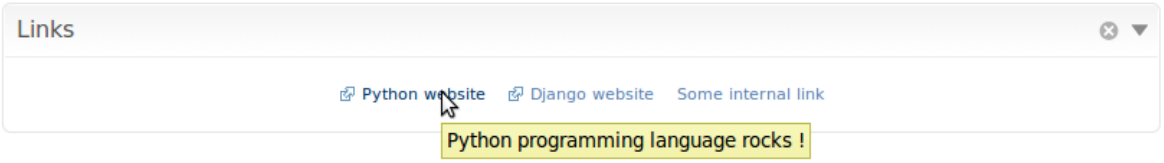
Children can also be iterables (lists or tuples) of length 2, 3, 4 or 5.

Here's a small example of building a link list module:

```
from admin_tools.dashboard import modules, Dashboard

class MyDashboard(Dashboard):
```

```
    def __init__(self, **kwargs):
        Dashboard.__init__(self, **kwargs)

        self.children.append(modules.LinkList(
            layout='inline',
            children=(
                {
                    'title': 'Python website',
                    'url': 'http://www.python.org',
                    'external': True,
                    'description': 'Python language rocks !',
                    'attrs': {'target': '_blank'},
                },
                ['Django', 'http://www.djangoproject.com', True],
                ['Some internal link', '/some/internal/link/'],
            )
        ))
```

The screenshot of what this code produces:



The `AppList` class

class admin_tools.dashboard.modules.**AppList**(*title=None*, ***kwargs*)

Module that lists installed apps and their models. As well as the *DashboardModule* properties, the *AppList* has two extra properties:

models A list of models to include, only models whose name (e.g. "blog.comments.models.Comment") match one of the strings (e.g. "blog.*") in the models list will appear in the dashboard module.

exclude A list of models to exclude, if a model name (e.g. "blog.comments.models.Comment") match an element of this list (e.g. "blog.comments.*") it won't appear in the dashboard module.

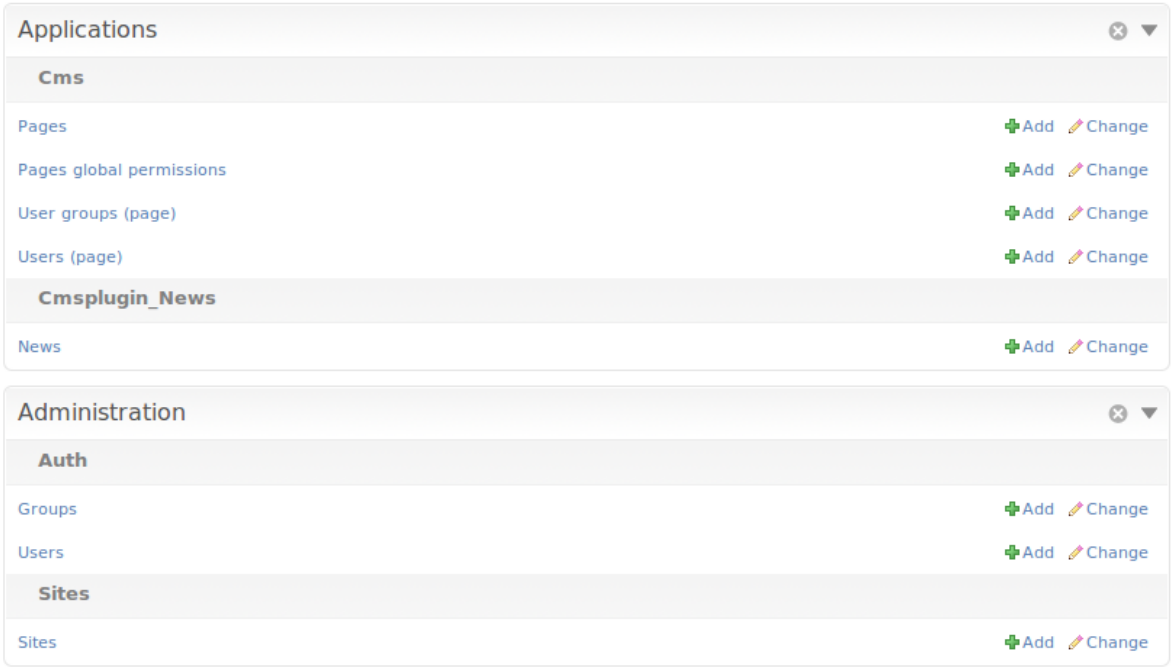
If no models/exclude list is provided, **all apps** are shown.

Here's a small example of building an app list module:

```
from admin_tools.dashboard import modules, Dashboard

class MyDashboard(Dashboard):
    def __init__(self, **kwargs):
        Dashboard.__init__(self, **kwargs)

        # will only list the django.contrib apps
        self.children.append(modules.AppList(
            title='Administration',
            models=('django.contrib.*',)
        ))
        # will list all apps except the django.contrib ones
        self.children.append(modules.AppList(
            title='Applications',
```

```
            exclude=('django.contrib.*',)
        ))
```

The screenshot of what this code produces:



Note: Note that this module takes into account user permissions, for example, if a user has no rights to change or add a `Group`, then the django.contrib.auth.Group model line will not be displayed.

The `ModelList` class

class `admin_tools.dashboard.modules.`**`ModelList`**(*title=None*, *models=None*, *exclude=None*, ***kwargs*)

Module that lists a set of models. As well as the *`DashboardModule`* properties, the *`ModelList`* takes two extra arguments:

`models` A list of models to include, only models whose name (e.g. "blog.comments.models.Comment") match one of the strings (e.g. "blog.*") in the models list will appear in the dashboard module.

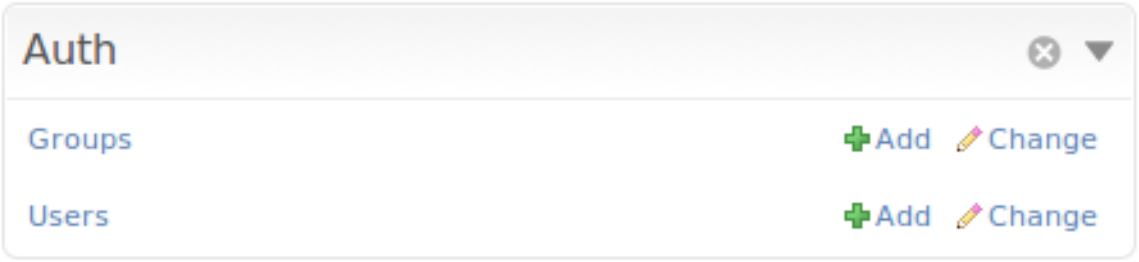
`exclude` A list of models to exclude, if a model name (e.g. "blog.comments.models.Comment") match an element of this list (e.g. "blog.comments.*") it won't appear in the dashboard module.

Here's a small example of building a model list module:

```
from admin_tools.dashboard import modules, Dashboard

class MyDashboard(Dashboard):
    def __init__(self, **kwargs):
        Dashboard.__init__(self, **kwargs)

        # will only list the django.contrib.auth models
        self.children += [
```

```
            modules.ModelList(
                title='Authentication',
                models=['django.contrib.auth.*',]
            )
        ]
```

The screenshot of what this code produces:



Note: Note that this module takes into account user permissions, for example, if a user has no rights to change or add a `Group`, then the django.contrib.auth.Group model line will not be displayed.

The `RecentActions` class

class admin_tools.dashboard.modules.**RecentActions**(*title=None*, *limit=10*, *include_list=None*, *exclude_list=None*, ***kwargs*)

Module that lists the recent actions for the current user. As well as the *DashboardModule* properties, the *RecentActions* takes three extra keyword arguments:

include_list A list of contenttypes (e.g. "auth.group" or "sites.site") to include, only recent actions that match the given contenttypes will be displayed.

exclude_list A list of contenttypes (e.g. "auth.group" or "sites.site") to exclude, recent actions that match the given contenttypes will not be displayed.

limit The maximum number of children to display. Default value: 10.

Here's a small example of building a recent actions module:

```
from admin_tools.dashboard import modules, Dashboard

class MyDashboard(Dashboard):
    def __init__(self, **kwargs):
        Dashboard.__init__(self, **kwargs)

        # will only list the django.contrib apps
        self.children.append(modules.RecentActions(
            title='Django CMS recent actions',
            include_list=('cms.page', 'cms.cmsplugin',)
        ))
```

The screenshot of what this code produces:

The `Feed` class

class `admin_tools.dashboard.modules.`**`Feed`**(*title=None*, *feed_url=None*, *limit=None*, ***kwargs*)
Class that represents a feed dashboard module.

Important: This class uses the Universal Feed Parser module to parse the feeds, so you'll need to install it, all feeds supported by FeedParser are thus supported by the Feed

---

As well as the *`DashboardModule`* properties, the *`Feed`* takes two extra keyword arguments:

`feed_url` The URL of the feed.

`limit` The maximum number of feed children to display. Default value: None, which means that all children are displayed.

Here's a small example of building a recent actions module:

```
from admin_tools.dashboard import modules, Dashboard

class MyDashboard(Dashboard):
    def __init__(self, **kwargs):
        Dashboard.__init__(self, **kwargs)

        # will only list the django.contrib apps
        self.children.append(modules.Feed(
            title=_('Latest Django News'),
            feed_url='http://www.djangoproject.com/rss/weblog/',
            limit=5
        ))
```

The screenshot of what this code produces:

CHAPTER 8

Integration with third party applications

todo: write doc for “Integration with third party applications” section.

CHAPTER 9

Contributing to django-admin-tools

You are very welcome to contribute to the project! django-admin-tools is on Github, which makes collaborating very easy.

There are various possibilities to get involved, for example you can:

- Report bugs, preferably with patches if you can
- Discuss new features ideas
- Fork the project, implement those features and send a pull request
- Enhance the documentation
- Translate django-admin-tools in your language

CHAPTER 10

Testing of django-admin-tools

This is information for developers of django-admin-tools itself.

Running tests

First, cd the test_proj directory:

```
$ cd test_proj
```

And to run the tests, just type:

```
$ python manage.py test
```

Code coverage report

Install the coverage.py library and the django-coverage app:

```
$ pip install coverage django-coverage
```

Then run tests and open test_proj/_coverage/index.html file in browser.

Where tests live

Unit tests should be put into appropriate module's tests.py. Functional/integration tests should be put somewhere into test_proj.

Python Module Index

a

Index