yawd-admin Documentation

Release 0.7.1-rc1

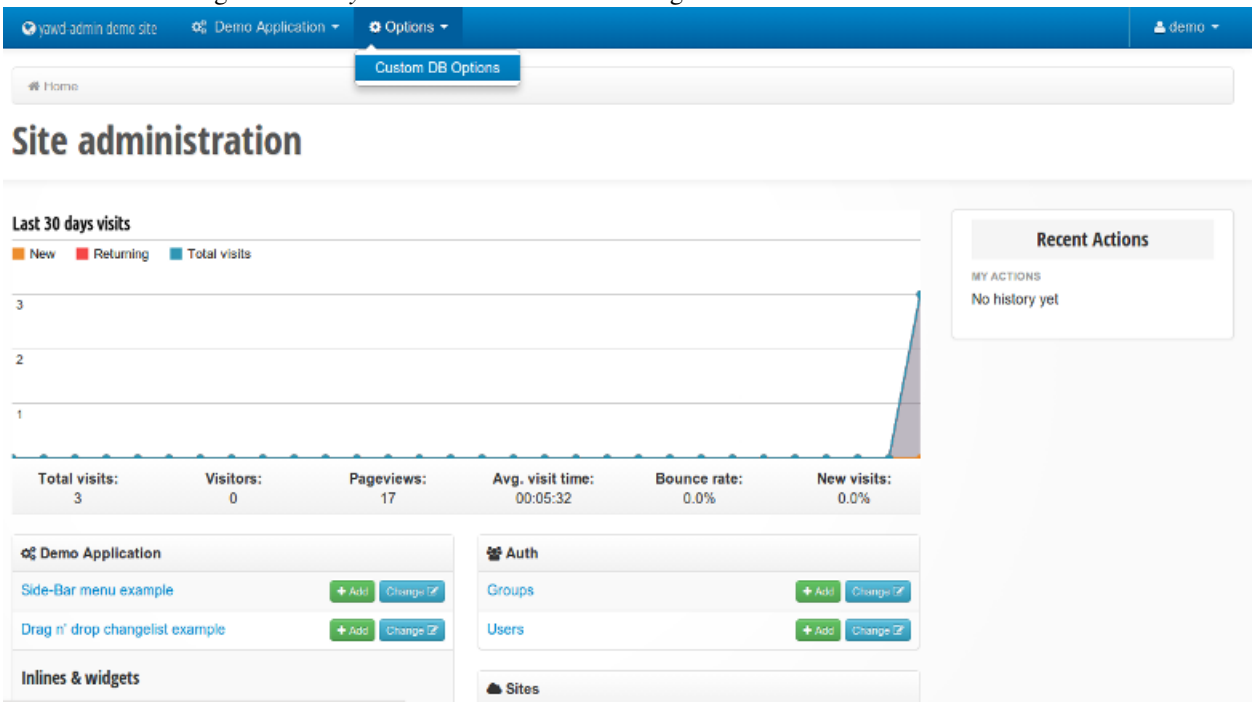
yawd

January 22, 2015

Contents

yawd-admin is an administration website for django. It extends the default django admin site and offers the following:

- A clean and beautiful bootstrap user interface
- Hand-written pure HTML5/CSS3 code with indented HTML output
- Responsive interface, optimized for mobile phones and tablets
- Register custom database settings (options) editable from the UI. You can use all **standard django form fields** for these settings
- Integration with google analytics for displaying statistics in the admin home page
- Register your applications to the top-bar navigation
- Refurbished original django admin widgets
- Mechanism for opening the original django admin popup windows with fancybox
- Seamless integration with *yawd-translations* for multilingual admin websites



Screenshot of the yawd-admin index page (taken from the http://yawd.eu/ production website).

CHAPTER 1

Django version compatibility

The following yawd-admin versions might also work with other Django releases (older or newer), however such combinations are NOT tested.

yawd-admin v0.5.0: Django v.1.4.1
yawd-admin v0.6.1: Django v.1.4.5
yawd-admin v0.7.0: Django >= v.1.5, < 1.6

CHAPTER 2

Contents

2.1 Installing yawd-admin

2.1.1 Prerequisites

yawd-admin requires that the *oauth2client* and *httplib2* libraries are installed in your python environment:

```
$ pip install oauth2client
$ pip install httplib2
```

If you plan to use the yawd-admin google analytics functionality, the *google-api-python-client* must also be installed:

```
$ pip install google-api-python-client
```

To see which Django versions work with yawd-admin please see *Django version compatibility*.

2.1.2 Application installation

Install yawd-admin either from pypi or the github repository:

```
$ pip install yawd-admin
```

...or:

```
$ git clone https://github.com/yawd/yawd-admin
$ cd yawd-admin
$ python setup.py install
```

Either ways all *Prerequisites* (**except** from *google-api-python-client*) will be automatically installed.

Note: Since yawd-admin is actively being developed the github version is generally preferred (especially if you use the latest django); the `master` branch always contains all latest updates & fixes and is generally considered stable.

2.1.3 Install the demo project

In the demo project you will find a full example of yawd-admin (only the google analytics functionality is not demonstrated).

Create a new environment named *yawdadmin* and activate it:

```
$ virtualenv /www/yawdadmin
$ source /www/yawdadmin/bin/activate
```

Download and install yawd-admin:

```
$ git clone https://github.com/yawd/yawd-admin
$ cd yawd-admin
$ python setup.py install
```

At this point, yawd-admin will be in your `PYTHONPATH`. Now initialize the example project:

```
$ cd example_project
$ python manage.py syncdb
```

When promted, create an admin account. Finally, start the web server:

```
$ python manage.py runserver
```

...and visit *http://localhost:8000/* to see the admin interface and experiment with its features.

Note: The demo admin URLs are registered in the root url and no prefix (e.g. '/admin/') is needed.

Explore the demo project source code and read the comments, to better understand how to use yawd-admin.

Once you are done, you can deactivate the virtual environment:

```
$ deactivate yawdadmin
```

2.1.4 Live demo

Alternatively, you can visit the live demo at http://yawd-admin.yawd.eu/. Use demo / demo as username & passowrd.

Note: The live demo uses yawd-admin v0.7.0.

2.2 Using yawd-admin

2.2.1 Set up

To use the yawd-admin website you first need to include yawdadmin in your `INSTALLED_APPS` setting. You also not need to include *django.contrib.admin* in `INSTALLED_APPS` and place it **after** *yawdadmin* for yawd-admin to work properly. In *settings.py*:

```
INSTALLED_APPS = (
        ...
        'yawdadmin',
        'django.contrib.admin',
        ...
)
```

yawd-admin uses the `yawdadmin.middleware.PopupMiddleware` middleware to replace the standard django admin popups with fancybox. Make sure the middleware is enabled in your `MIDDLEWARE_CLASSES` setting. In *settings.py*:

```
MIDDLEWARE_CLASSES = (
        ...
        'yawdadmin.middleware.PopupMiddleware',
        ...
)
```

Note:

```
If you're running Django 1.6 and above, you don't need to use this middleware.
```

Finally, the *django.core.context_processors.request* context processor must also be enabled:

```
TEMPLATE_CONTEXT_PROCESSORS = (
        ...
        "django.core.context_processors.request",
        ...
)
```

2.2.2 Registering the yawd-admin urls

To register the admin site views, use the following (inside your *urls.py*):

```
from yawdadmin import admin_site

patterns = ('')
        url(r'^admin/', include(admin_site.urls)),
        ...
)
```

You **do not** need to register the django admin urls as well, the yawd `admin_site` extends the original admin class.

2.2.3 ModelAdmin registration and auto-discovery

Normally, to register your normal ModelAdmin class with yawd-admin you should use `yawdadmin.admin_site` instead of the original `django.contrib.admin.site` instance (in *admin.py*):

```
from django.contrib import admin
from models import MyModel

class MyModelAdmin(admin.ModelAdmin):
        pass

from yawdadmin import admin_site
#you can use this instead of admin.site.register():
admin_site.register(MyModel, MyModelAdmin)
```

However, many applications might have registered their *ModelAdmin* classes with the default django admin site. As you can see from the above snippet yawd-admin uses the *ModelAdmin* class as well, therefore you can easily add all standard registrations to the yawd-admin website. To do so, use the standard *admin.autodiscover()* method and then update the yawd-admin registry as follows (in *urls.py*):

```
from django.contrib import admin
from yawdadmin import admin_site

admin.autodiscover()
admin_site._registry.update(admin.site._registry)
```

2.2.4 Integration with Custom User Models

yawdadmin comes with a new admin view to allow staff users edit their own account information (username, first name, last name and email). This view uses a ModelForm of the standard `django.contrib.auth.models.User` model.

If your projects makes use of the new django 1.5 custom user functionality you can set the `ADMIN_USER_MODELFORM` yawd-admin setting to override the ModelForm used by the view (in settings.py):

```
ADMIN_USER_MODELFORM = 'myapp.module.MyModelForm'
```

Note that the setting value can be a string or Class. A string is normally preferred to avoid import errors during environment initialization.

2.2.5 Settings

ADMIN_DISABLE_APP_INDEX

With yawd-admin you can optionaly disable the app index view (the one that lists an application's models). Doing so will raise "Page Not Found" (404) errors when accessing the application urls and will also hide all corresponding links from breadcrumbs.

```
ADMIN_DISABLE_APP_INDEX = True
```

ADMIN_GOOGLE_ANALYTICS

A dictionary holding configuration of the connected google analytics account. Please see *Integration with Google Analytics*.

ADMIN_SITE_NAME / ADMIN_SITE_DESCRIPTION

You can change the admin site name and add a description to the login page by adding a couple attributes to your settings:

```
ADMIN_SITE_NAME = 'My Admin Site'
ADMIN_SITE_DESCRIPTION = 'This is a private site.  Please don\'t hack me'
```

If you don't want a description at all just null the attribute:

```
ADMIN_SITE_DESCRIPTION = None
```

ADMIN_SITE_LOGO_HTML

To set a logo that will show up in the right side of the header:

```
ADMIN_SITE_LOGO_HTML = '<div id="myproject-logo hidden-phone">Logo</div>'
```

ADMIN_JS_CATALOG

Additional javascript translation messages for use in the admin interface. Please see *Javascript message translations*.

ADMIN_JS_CATALOG_CACHE_TIMEOUT

For how long you want the js catalog view to be cached. Yawd-admin will cache a different version of the view per language. Use 0 if you want to disable caching on the jsi18n view. Defaults to 60 * 60 (1 hour).

ADMIN_USER_MODELFORM

If you implement a custom user model (django 1.5 and above) you can override the ModelForm that yawd-admin uses to allow staff users edit their account data. For more info please see *Integration with Custom User Models*.

```
ADMIN_USER_MODELFORM = 'myapp.module.MyModelForm'
```

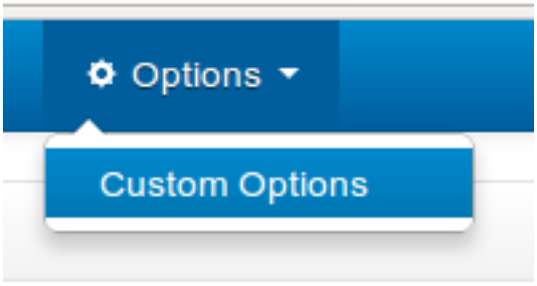
ADMIN_CACHE_DB_OPTIONS

default: *3600*

Cache duration (in seconds) for the admin database options. Cache gets updated each time an option value changes. That way cached values always reflect the db values. If set to 0 there will be no caching.

2.3 DB options

With yawd-admin you can register sets of custom options. These options are editable through the admin interface and you can use them to let administrators fine-tune the application.



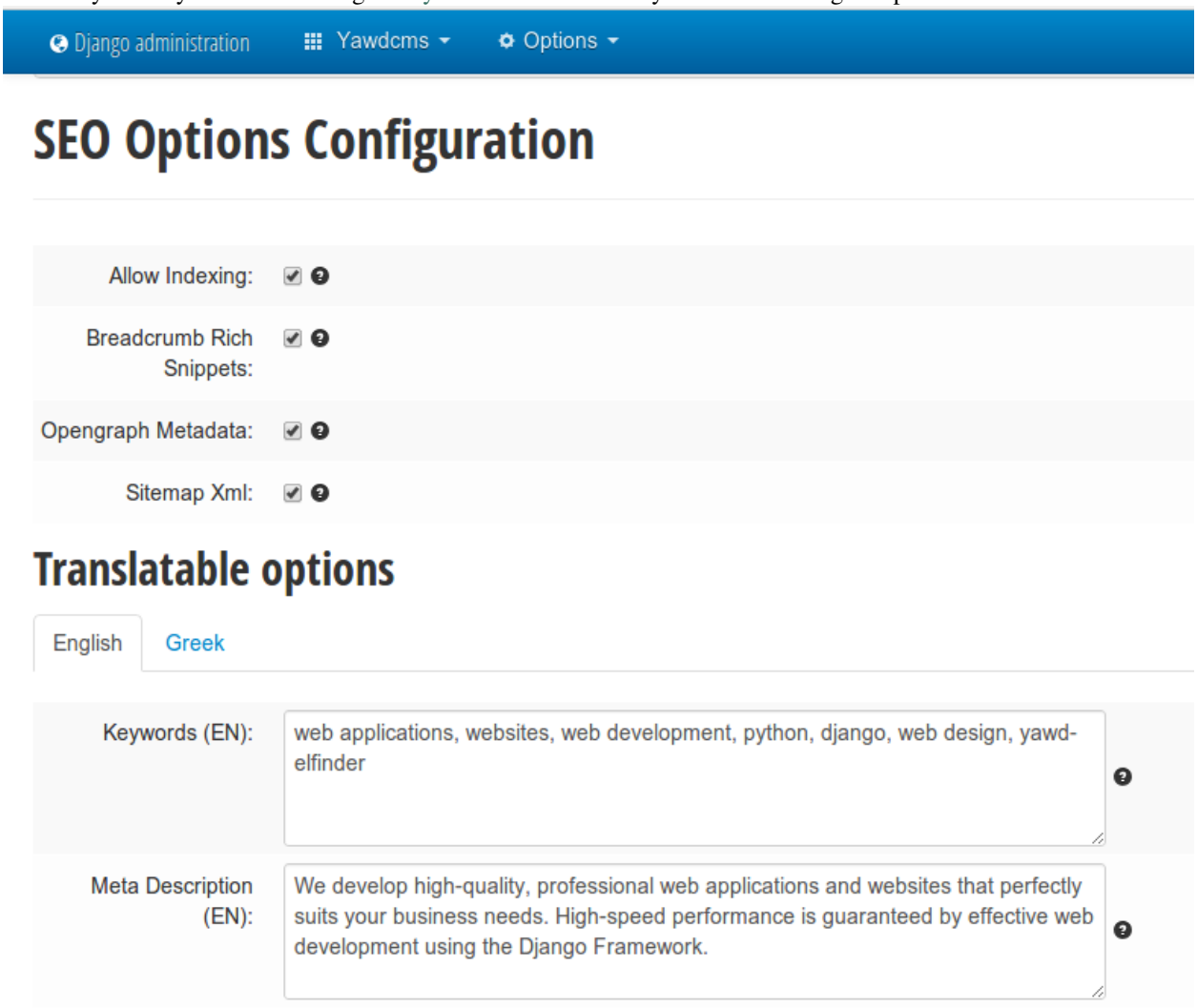
Each set of options is defined by extending the `yawdadmin.admin_options.OptionSetAdmin` class:

```
class CustomOptions(OptionSetAdmin):
        optionset_label = 'custom-options'
        verbose_name = 'Custom Options'

        option_1 = SiteOption(field=forms.CharField(
                widget=forms.Textarea(
                        attrs = {'class' : 'textarea-medium'}
                ),
                required=False,
                help_text='A fancy custom text area option.',
        ))

        option_2 = SiteOption(field=forms.CharField(
                help_text='The second awesome option. This one is required!',
        ))
```

The `optionset_label` attribute is the equivalent of the `app_label` for models. By defining a `verbose_name` you can explicitly set how you want this option-set label to be displayed.

Each option is implemented as a member of the `OptionSetAdmin` sub-class, exactly like you would do in a database model. The options must be of the `yawdadmin.admin_options.SiteOption` type. The `field` argument of the *SiteOption* constructor can refer to any standard django form field class instance. In the above example, *option_1* will be a text area and *option_2* a text input.

Note: a *SiteOption* initialization can accept a `lang_dependant` boolean keyword argument as well. Set this to `True` if you use yawd-admin along with yawd-translations and you need multilingual options:



After defining your custom `OptionSetAdmin` class you must register it with the yawd-admin website:

```
#register the OptionSetAdmin to the admin site
#almost like we would do for a ModelAdmin
admin_site.register_options(CustomOptions)
```

2.3.1 Retrieving option values

To retrieve a single option you can use the `get_option()` method:

```
from yawdadmin.utils import get_option
option = get_option('custom-options', 'option_1')

if option == 'whatever value':
        #do your stuff..
```

... where the first argument of the method is the *optionset_label* and the second is the option name.

If you want to retrieve all options of a single option-set at once use the `get_options()` method (if you need access to more than one options this is preferred since it will hit the database only once):

```
from yawdadmin.utils import get_options
options = get_options('custom-options')

if options['option_1'] == 'whatever value':
        #do your stuff
```

...or in the template:

```
<p><span>Option 1 value:</span> {{options.option_1}}</p>
```

2.4 Inlines

2.4.1 Admin inline customizations

Collapsing inlines

With yawd-admin you can collapse your inlines, like you do with your fieldsets. Collapsing an admin inline is easy and works for both stacked and tabular inlines:

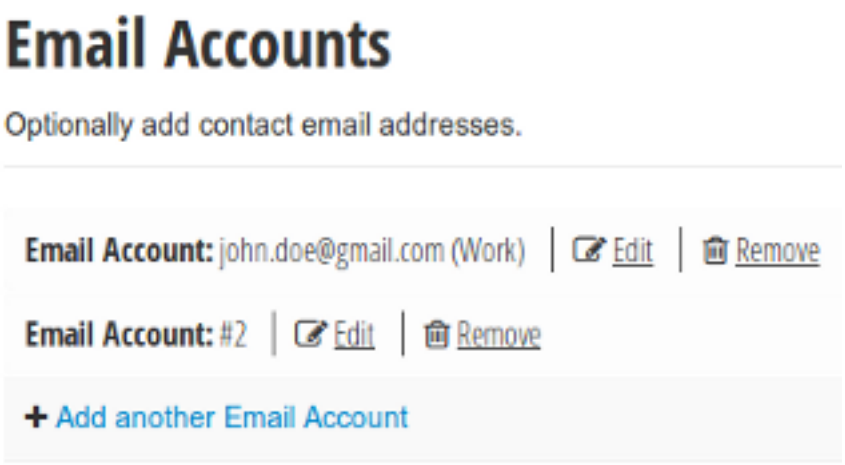
```
class MyStackedInline(admin.StackedInline):
        ...
        collapse = True

class MyTabularInline(admin.TabularInline):
        ...
        collapse = True
```

Inline description

When setting a model's fieldsets you can provide a `description` key to specify a text that will be displayed under the fieldset header. Now you can achieve the same effect with your inlines using the `description` member in your Inline class:

```
class MyStackedInline(admin.StackedInline):
        ...
        description = 'My inline description text'
```

Modal inlines



Another nice option is the inline modal functionality. It can be really useful when you have a lot of fields in your inline model. Add `modal=True` to the `StackedInline` class and your inline form will open in a popup-style modal window:

```
class MyStackedInline(admin.StackedInline):
        ...
        modal = True
```

This does not work with tabular inlines

2.4.2 New inline types

In addition to the stacked and tabular inlines, yawd-admin implements a couple of new inline types.

Popup/Ajax inlines

This inline loads the inline form using ajax. It comes in handy when you want to have inline forms *inside* an inline, or you jusst want a better-looking inline form.

Say we want to implement an image gallery where each image should have variations. An example models.py could look like this:

```
class Gallery(models.Model):
        title = models.CharField(max_length=100, required=True)

class GalleryImage(models.Model):
        gallery = models.ForeignKey(Gallery)
        title = models.CharField(max_length=100)

class GalleryImageVariation(models.Model)
        image = models.ForeignKey(GalleryImage)
        image_file = models.ImageField(upload_to=...)
```

We intend to use GalleryImageVariation as an inline for GalleryImage. In admin.py:

```
class GalleryAdmin(admin.ModelAdmin):
        ...

class GalleryImageVariationInline(models.StackedInline):
        model = GalleryImageVariation
        ....

class GalleryImageAdmin(admin.ModelAdmin):
        inlines = (GalleryImageVariationInline,)
        ...

admin_site.register(Gallery, GalleryAdmin)
admin_site.register(GalleryImage, GalleryImageAdmin)
```

Now what if we wanted images to be editable through the Gallery admin page? We could use a StackedInline or a Tabular inline for the GalleryImage model but still all ImageVariations would be editable only through the GalleryImage's own change page. To solve this we can use the `PopupInline`, as follows:

```
from yawdadmin.admin import PopupInline, PopupModelAdmin

class GalleryImageInline(PopupInline):
        model = GalleryImage
        ...

class GalleryAdmin(admin.ModelAdmin):
        inlines = (GalleryImageInline,)
        ...

class GalleryImageVariationInline(models.StackedInline):
        model = GalleryImageVariation
        ....

#We extend PopupModelAdmin instead of the original ModelAdmin.
#This adds a couple of extra views to the GalleryImageAdmin
#that allow managing the GaleryImage through ajax.
#We also need to explicitly set the linked inline class
#that will use the ajax functionality
class GalleryImageAdmin(PopupModelAdmin):
        inlines = (GalleryImageVariationInline,)
        linked_inline = GalleryImageInline
        ...

admin_site.register(Gallery, GalleryAdmin)
admin_site.register(GalleryImage, GalleryImageAdmin)
```

The above code will allow adding/editing images directly through the gallery change page. Each time you choose

to edit/add a gallery image, a modal form will open; this form actually is the original GalleryImageAdmin form, including all of its fieldsets, inlines etc.

Popup inlines can also be sortable. This means you can change their order using drag & drop:

```
class GalleryImage(models.Model):
        ...
        ordering_field = models.IntegerField(..)

class GalleryImageInline(PopupInline):
        model = GalleryImage
        sortable = False
        #the default value for the following property is 'order'.
        sortable_order_field = 'ordering_field'
        ...
```

If you want to allow editing gallery images only through the popup inline you can use the *popup_only* attribute:

```
class GalleryImageAdmin(PopupModelAdmin):
        inlines = (GalleryImageVariationInline,)
        linked_inline = GalleryImageInline
        popup_only = True
        ....

admin_site.register(GalleryImage, GalleryImageAdmin)
```

The above will raise `Http404` errors when accessing the standard add/change views of the `GalleryImage` model. However, it will not make the equivalent links disappear from the admin homepage or the app index page etc. To do so you must override the equivalent templates. For the top-bar navigation you can use the *Model exclusion* functionality.

Note: Ajax/Popup inlines are **only** displayed in the *change* form and not the *add* form of an object, since they require the parent object to be already saved in the database.

One-to-one inlines

If you have one-to-one relations, you might want to use the `OneToOneInline` class to make the add/change page look like that of a single model.

For example, say you're developing an e-shop selling books. In your admin.py:

```
from yawdadmin.admin import OneToOneInline

class BookInlineAdmin(OneToOneIneline):
        model = Book
        #by default show_title is True
        #disable show_title if you want to to hide the model verbose name
        #and use it in conjuction with fieldsets to achieve the desired effect
        show_title = False

class ProductAdmin(admin.ModelAdmin):
        inlines = (BookInline,)
        ....

admin_site.register(Product, ProductAdmin)
```

That way the `Product` admin pages will look like a single form is being used.

2.5 Integration with Google Analytics

To access your google analytics reports through the yawd-admin index page you need to first create a new google API application by performing the following steps:

- Visit the Google APIs Console (https://code.google.com/apis/console)
- Sign-in and create a project or use an existing project.
- In the Services pane (https://code.google.com/apis/console#:services) activate Analytics API for your project. If prompted, read and accept the terms of service.
- Go to the API Access pane (https://code.google.com/apis/console/#:access):
- Click Create an OAuth 2.0 client ID:
  - Fill out the Branding Information fields and click Next.
  - In Client ID Settings, set Application type to 'Web application'.
  - In the **Your site or hostname** section click 'more options'.
    - The **Authorized redirect URIs** field must be set to `http://localhost:8000/admin/oauth2callback/` (trailing slash seems to matter at this point of time). Replace *localhost:8000* with a domain if you are on a production system. The '/admin/' part of the URL refers to the *prefix* you used to register the admin site with.
    - The **Authorized JavaScript Origins** field must be set to `http://localhost:8000/` (or the domain root if you are on a production system).
  - Click Create client ID

Keep a node of the generated *Client ID* and *Client secret* as we will use them later on.

Go into your project source files and create a new file named *client_secrets.json*. The file contents should look like this:

```
{
  "web": {
    "client_id": "[[INSERT CLIENT ID HERE]]",
    "client_secret": "[[INSERT CLIENT SECRET HERE]]",
    "redirect_uris": [],
    "auth_uri": "https://accounts.google.com/o/oauth2/auth",
    "token_uri": "https://accounts.google.com/o/oauth2/token"
  }
}
```

Replace `[[INSERT CLIENT ID HERE]]` and `[[INSERT CLIENT SECRET HERE]]` with the actual *Client ID* and *Client secret* you created in the previous step.

Now all we need to do is enable the google analytics in the project settings module (*settings.py*):

```
ADMIN_GOOGLE_ANALYTICS = {
        'client_secrets' : '/absolute/path/to/client_secrets.json',
        'token_file_name' : '/absolute/path/to/analytics.dat',
        'profile_id' : '12345678',
        'admin_root_url' : 'http://localhost:8000/admin/'
}
```

The `client_secrets` key must hold the absolute path to the the *client_secrets.json* file we created.

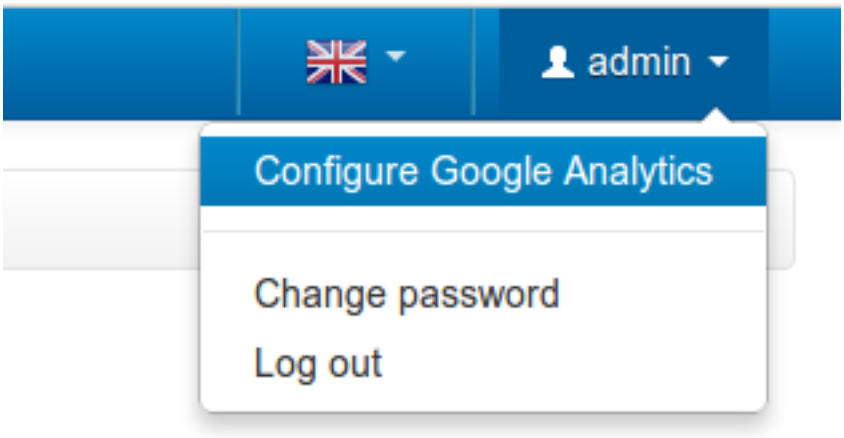
The `token_file_name` key must point to the absolute path of a file where yawd-admin will store session keys and information returned from the google API. You do not need to manually create this file, just make sure the web server has write access to that path.

`profile_id` refers to the ID of the google analytics account you want to connect with yawd-admin. To find this ID login to your google analytics account, click the 'Admin' link from the horizontal menu and select the account you wish to connect.

A screenshot of an analytics account showing the Profile ID.

The last setting, `admin_root_url` must be set to the root url of the admin website.

Now restart the web server and visit the admin interface (e.g. http://localhost:8000/admin).

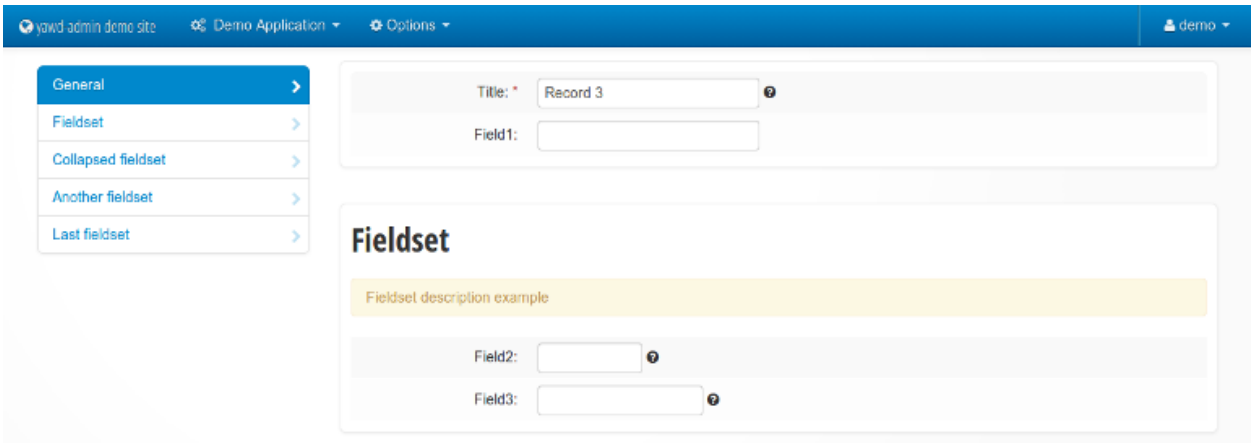


Visit the 'Configure Google Analytics' page (image above) and click 'Authenticate new account' to grant the application access to your google analytics data. Make sure the google account you link has access to the specified `profile_id`.

Now yawd-admin has stored your data and you don't need to go through the confirmation process again.

2.6 Additional functionality

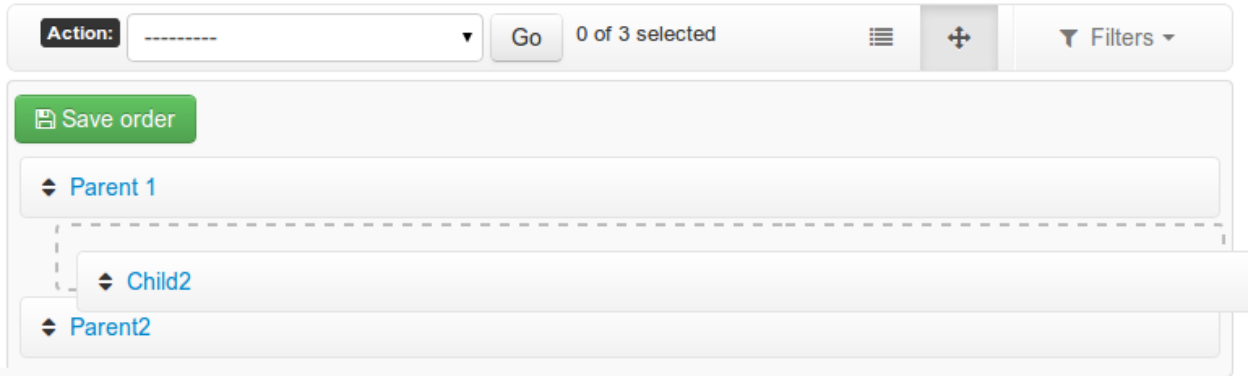
2.6.1 Side navigation for change forms



You can optionally enable a left menu navigation for your change form pages on any model. This will automatically list and track all fieldsets and inlines set in the ModelAdmin:

```
class MyModelAdmin(admin.ModelAdmin):
        ..other stuff..
        fieldsets = (...)
        inlines = (...)
        affix=True
```

2.6.2 Sortable changelists



You can enable a "sorting mode" in the changelist view for orderable objects by subklassing `yawdadmin.admin.SortableModelAdmin` instead of `admin.ModelAdmin`:

```
#Model admin class
class CategoryAdmin(SortableModelAdmin):
        ...
        ...

admin_site.register(Category, CategoryAdmin)
```

By default yawdadmin expects the ordering model field to be named "order" (it must be an *IntegerField*). If the name is different you need to set the *"sorting_order_field"* attribute:

```
#model definition
class Category(models.Model):
        ...
        weird_order_field_name = models.IntegerField(default=0)

#Model admin class
class CategoryAdmin(SortableModelAdmin):
        sortable_order_field = 'weird_order_field_name'
        ...
```

If you use django-mptt for nested categories, you can enable nested ordering like so (see screenshot above):

```
#Model admin class
class CategoryAdmin(SortableModelAdmin):
        sortable_mptt = True
        ...
```

The sorting mechanism assumes items are orderd by the ordering field in the default queryset. If that's not true, you should override the *"sortables_ordered"* method to provide a proper default ordering:

```
#Model admin class
class CategoryAdmin(SortableModelAdmin):
        def sortables_ordered(self, queryset):
                return queryset.order_by("order")
```

2.6.3 Javascript message translations

If you use *custom templates* and want to add multilingual javascript messages as described in the django documentation, you can use the *ADMIN_JS_CATALOG* setting:

```
ADMIN_JS_CATALOG = ['your.app.package', 'your.app.package.2']
```

Make sure you have compiled the translated javascript messages (*djangojs* namespace) for all listed applications, so they're included in the admin catalog view.

2.6.4 Model icons

You can set an accompanying icon class for each of your models in the `ModelAdmin` class.

```
class MyModelAdmin(admin.ModelAdmin):
        ....
        title_icon = 'icon-group'
```

yawd-admin will display this icon in various places (e.g drop-down menus, change list pages, change form pages) in an effort to make your UI more eye-appealing.

The icon classes you can choose from are listed here. yawd-admin uses the font-awesome bootstrap icons instead of the original ones. Therefore you can apply any CSS rule to customize the look & feel of your icons.

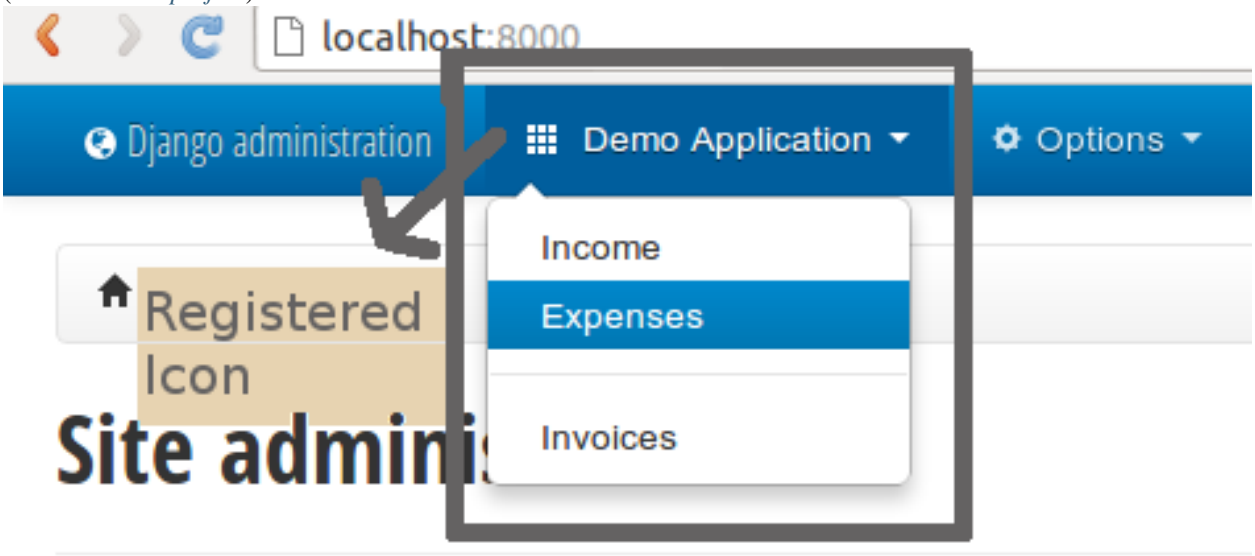
2.7 Top-bar navigation

yawd-admin provides a top navigation bar. If you wish, you can register an application's admin models along with an accompanying image to the top-bar as follows:

```
from yawdadmin import admin_site
admin_site.register_top_menu_item('sites', icon_class="icon-th")
```

The `icon_class` argument can be any icon from the ones that ship with bootstrap, found here.

The above snippet will register the *django.contrib.admin.sites* application to the top bar. Note however that if the application you try to register is not yet registered with the admin website, an Exception will be raised. Therefore, a safe place to put this code is in your *urls.py module*, right after the *auto-discovery* code. If you want to register the current application, you could use the *admin.py* module and place the code right after the *ModelAdmin* registrations (as in the *demo project*).



A screenshot of the top-bar navigation from the demo project. Note that the order in which *ModelAdmin* classes are presented in the drop-down box is not alphabetical and that there is also a separator line between the *Expenses* and *Invoices* items. yawd-admin provides two custom *ModelAdmin* attributes to achieve this behavior: `order` and `separator`. You can use them like this:

```
class MyCategoryAdmin(admin.ModelAdmin)
        ... #bla bla..
        order = 2

class MyPageAdmin(admin.ModelAdmin)
        ... #bla bla..
        order = 1

class MyThirdAdmin(admin.ModelAdmin)
        ... #bla bla..
        order = 3
        separator = True
```

The above will place *MyPageAdmin* before *MyCategoryAdmin* and *MyThirdAdmin* will come last. A separator line will also be drawed **before** the *MyThirdAdmin* item.

If you do not set a custom *ModelAdmin* order, yawd-admin will use the standard alphabetical order for your models.

2.7.1 Model exclusion

You can exclude a certain model from the top-bar navigation. To do so set the `exclude_from_top_menu` attribute to True:

```
class MyExcludedAdmin(admin.ModelAdmin)
        ... #bla bla..
        exclude_from_top_menu = True
```

Custom top-bar menus

In addition to the app/model-driven top bar menus, you can also create custom menus. To do that you should use the `register_top_menu_item` method, specifying child menu items like this:

```
from yawdadmin import admin_site
admin_site.register_top_menu_item('Custom menu', icon_class="icon-th",
        children=[{'name': 'Custom view 1', 'admin_url': reverse_lazy('custom-url-view'), 'order': 1,
                  {'name': 'Custom view 2', 'admin_url': reverse_lazy('custom-url-view-2'), 'order':
perms=perms_func)
```

The `children` keyword argument must be a list holding the actual sub-menu items. Each item in this list must be a dictionary with the following keys:

- *name*: The menu item name. **Required**
- *admin_url*: The menu item URL. **Required**
- *title_icon*: The class of the leading icon. Optional.
- *order*: The item's order among its siblings. Optional.
- *separator*: If a separator should be placed *before* this item (just like with model-driven menus). Optional.

The `perms` keyword argument is **optional**. If you wish to control the permissions on each menu item, you can specify a function that accepts both the current request and a menu item as arguments and returns either True -when the user is allowed to view the item-, or False. Example implementation:

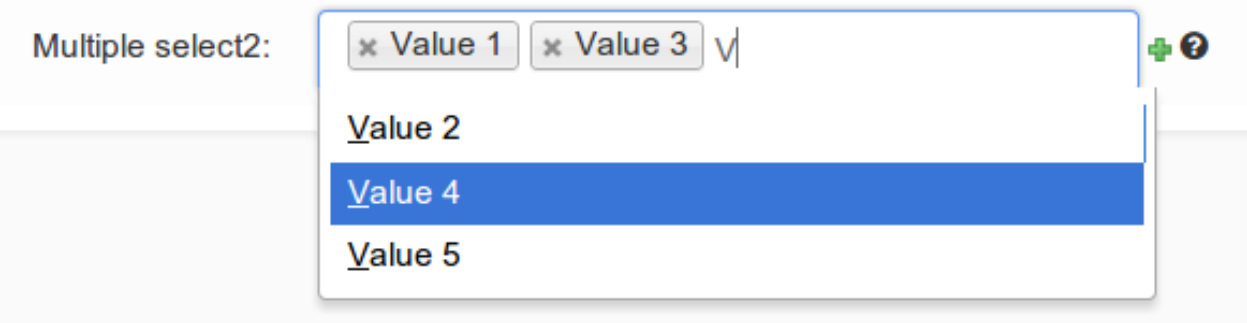
```
def perms_func(request, item):
        if not request.user.is_superuser and item['admin_url'].startswith('/private'):
                return False
        return True
```

2.8 Widgets

yawd-admin implements some admin widgets for use in your applications.

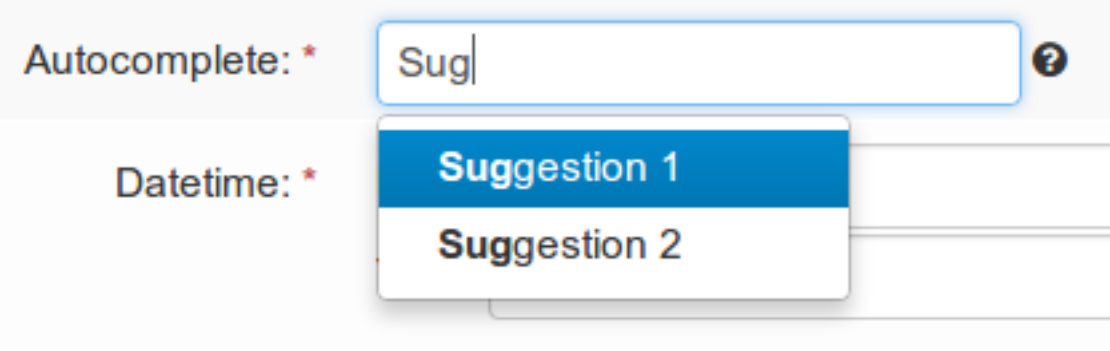
2.8.1 Select2MultipleWidget widget

You can use this widget instead of the default *SelectMultipleWidget* for a prettier multiple choice selection input.



```
from yawdadmin.widgets import Select2MultipleWidget

class MyForm(forms.ModelForm):
    class Meta:
        widgets = {
            'multiplefield': Select2MultipleWidget
        }
```

2.8.2 AutoCompleteTextInput widget



yawd-admin implements a bootstrap typeahead widget that you can use in your forms. As you type in the text input, the widget will provide suggestions for auto-completing the field.

Say for example there is a `Contact` model having a field named `profession`. You want the *profession* text input to suggest professions while typing. First you should create a view that returns a json-serialized object with the suggestions:

```
class TypeaheadProfessionsView(View):
    def get(self, request, *args, **kwargs):
        if not request.is_ajax():
        raise PermissionDenied

        query = request.GET.get('query', None)
        results = []

        for el in Contact.objects.values_list('profession', flat=True).distinct():
            if el and (not query or el.find(query.decode('utf-8')) != -1):
                results.append(el)

        return HttpResponse(json.dumps({'results': results}))
```

As you type in the text field, the js code makes a get request to your custom view, with the typed text being sent in the `query` *GET variable*. As you can see from the code above, the dictionary returned by the view must have a `results` element that contains a list with all suggestions.

No suppose we register this view with the name *'profession-suggestions-view'*. We can create a custom admin form for the `Contact` and override the widget for the `profession` field as follows:

```
from yawdadmin.widgets import AutoCompleteTextInput

class MyContactForm(forms.ModelForm):
    class Meta:
        widgets = {
```

```
            'profession': AutoCompleteTextInput(source=reverse_lazy('profession-suggestions-view'))
        }
```

Finally, in our admin.py we must force the `Contact`'s model admin to use the custom form:

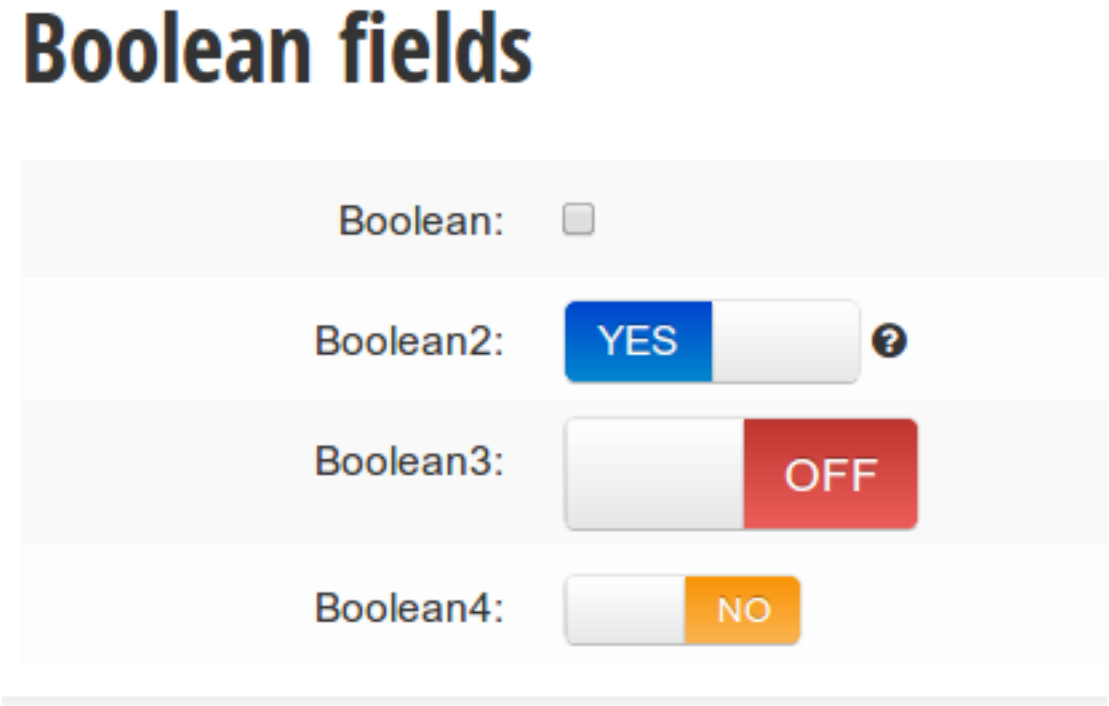
```
class MyContactAdmin(admin.ModelAdmin)
        form = MyContactForm
```

2.8.3 Radio buttons

To use the built-in bootstrap-style radio buttons use the `BootstrapRadioRenderer` renderer:

```
from yawdadmin.widgets import BootstrapRadioRenderer

class MyForm(forms.ModelForm):
    class Meta:
        widgets = {
            'myselectfield': forms.RadioSelect(renderer=BootstrapRadioRenderer)
        }
```

2.8.4 SwitchWidget



The *SwitchWidget* can be used on Boolean Fields to display smartphone-style switches instead of checkboxes.

```
from yawdadmin.widgets import SwitchWidget

class MyForm(forms.ModelForm):
    class Meta:
        widgets = {
```

```
            'booleanfield': SwitchWidget
        }
```

To change the button size use one of the following classes:

- switch-large
- switch-small
- switch-mini

```
from yawdadmin.widgets import SwitchWidget

class MyForm(forms.ModelForm):
    class Meta:
        widgets = {
            'booleanfield': SwitchWidget(attrs={'class': 'switch-small'})
        }
```

By default the switch uses the labels *YES*/*NO* for on and off positions. You can override the label text using the *data-on-label* and *data-off-label* attributes respectively:

```
from yawdadmin.widgets import SwitchWidget

class MyForm(forms.ModelForm):
    class Meta:
        widgets = {
            'booleanfield': SwitchWidget(attrs={'data-on-label': 'I\'M ON',
                        'data-off-label': 'I\'M OFF',})
           }
```

You can also change the color of the on and of positions using the *data-on* and *data-off* attributes. The available color tokens are the following:

- primary
- info
- warning
- success
- danger
- default

```
from yawdadmin.widgets import SwitchWidget

class MyForm(forms.ModelForm):
    class Meta:
        widgets = {
            'booleanfield': SwitchWidget(attrs={'data-on': 'success',
                                'data-off': 'danger',})
           }
```

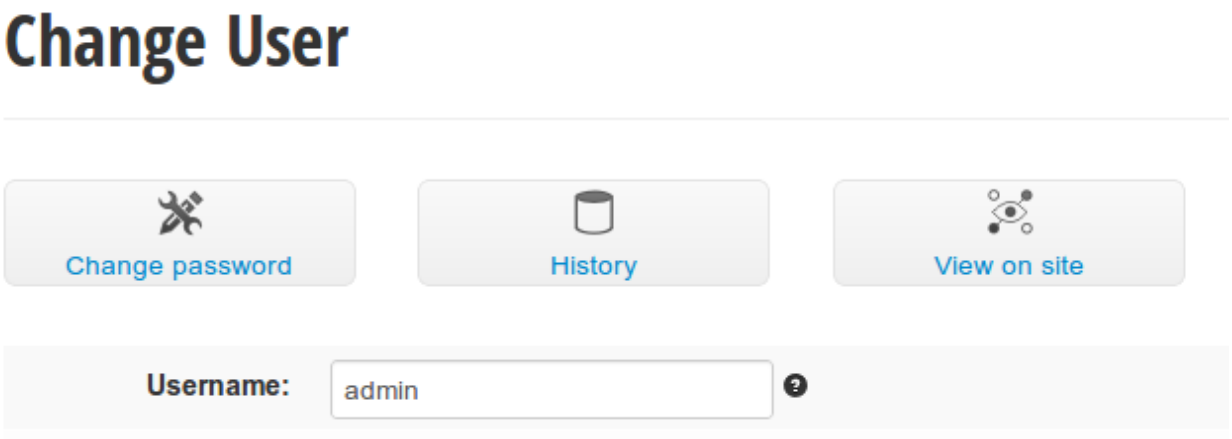
2.9 Templates

2.9.1 Overriding the templates

As with the default django admin site you can override the templates of yawd-admin. Just make sure your application is listed before 'yawd-admin' in your `INSTALLED_APPS` settings and place your templates inside the *myapp/templates/admin* directory. Per-model template override works exactly like the original django admin as well.

Object tools

The object tools in yawd-admin look like in the following screenshot (taken from the `User` model change page):



The html in templates for object tools is like this:

```
{% block object-tools-items %}
<div class="span2">
        <a href="history/" class="big-button">
                <i class="database-item"></i> {% trans "History" %}
        </a>
</div>{% if has_absolute_url %}
<div class="span2">
        <a href="../../../r/{{ content_type_id }}/{{ object_id }}/" class="big-button">
                <i class="lookup-item"></i> {% trans "View on site" %}
        </a>
</div>{% endif %}{% endblock %}
```

Since you might override the *'object-tools-items'* template block to add your custom object tools, yawd-admin provides a set of icons to use for your buttons.

To use the above icons you must use an an `{{iconname}}-item` class where `{{iconname}}` is the name of the icon. E.g.:

```
<i class="male-item"></i>
<i class="mobile-item"></i>
<i class="plane-item"></i>
```

Some icon names have an *'-item'* suffix on their own, you should leave the icon name as is in this case. E.g.:

```
<i class="add-item"></i>
<i class="copy-item"></i>
```

2.9.2 Templates for popular django applications

yawd-admin comes with templates for the following popular django applications:

```
* django-reversion (thanks `pahaz <https://github.com/pahaz>`_)
* django-mptt (thanks `pahaz <https://github.com/pahaz>`_)
* django-import-export
```

Just remember to place yawd-admin above these applications in your `settings.py` file.

2.10 Changelog

2.10.1 v.0.7.0, 2013.10.23

- popup/ajax inlines, one-to-one inlines
- auto side-navigation affix menu in change pages (based on fieldsets & inlines)
- drag & drop ordering in changelist view
- "my account" view for non-admin users to edit their own info
- new admin widgets
- reworked filters section
- reworked authentication/registration templates
- added the font-awesome icons (updated to 3.2.1)
- model icons
- automatic calculation of the popover (help text) placement
- documentation enhancements
- some work towards PEP8/pylint compliance

2.10.2 v.0.6.0, 2013.02.28

- Added modal inlines functionality
- Upgraded to bootstrap v2.2.2
- Added gcons icon set
- Collapsible inlines
- Fixed google analytics bugs (thanks to @chielteuben)
- Various css updates
- Added django-reversion, django-mptt templates (thanks to @pahaz)
- Various minor enhancements, like the ability to exclude a model from the top menu

2.10.3 v.0.5.0, 2012.11.01

- Initial Release

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*