Django Grappelli Documentation

Release 2.13.3

Patrick Kranzlmueller

Dec 27, 2019

Contents

A jazzy skin for the Django admin interface.

This documentation covers version 2.13.3 of Grappelli. Grappelli is a grid-based alternative/extension to the Django administration interface.

Note: Grappelli 2.13.3 requires Django 2.2. More on *Versions and Compatibility*.

CHAPTER 1

Installation & Setup

1.1 Quick start guide

For using Grappelli 2.13.3, Django 2.2 needs to be installed and an Admin Site has to be activated.

1.1.1 Installation

```
$ pip install django-grappelli
```

Go to https://github.com/sehmaschine/django-grappelli if you need to download a package or clone/fork the repository.

1.1.2 Setup

Open `settings.py` and add `grappelli` to your `INSTALLED_APPS` (before `django.contrib.admin`):

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

Add URL-patterns. The grappelli URLs are needed for related–lookups and autocompletes. Your admin interface is available with the URL you defined for `admin.site`:

```
urlpatterns = [
    path('grappelli/', include('grappelli.urls')), # grappelli URLS
    path('admin/', admin.site.urls), # admin site
]
```

Add the request context processor (needed for the Dashboard and the Switch User feature):

```
TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'django.template.context_processors.request',
                ...
            ],
        },
    },
]
```

Collect the media files:

```
$ python manage.py collectstatic
```

1.1.3 Testing

Start the devserver and login to your admin site:

```
$ python manage.py runserver <IP-address>:8000
```

Check if everything looks/works as expected. If you're having problems, see *Troubleshooting*.

1.2 Customization

While Grappelli is mainly about the look & feel of the admin interface, it also adds some features.

1.2.1 Available Settings

GRAPPELLI_ADMIN_TITLE The Site Title of your admin interface. Change this instead of changing index.html.

GRAPPELLI_AUTOCOMPLETE_LIMIT Number of items to show with autocomplete drop–downs.

GRAPPELLI_AUTOCOMPLETE_SEARCH_FIELDS A dictionary containing search patterns for models you cannot (or should not) alter.

GRAPPELLI_SWITCH_USER Set to `True` if you want to activate the switch user functionality.

GRAPPELLI_SWITCH_USER_ORIGINAL A function which defines if a User is able to switch to another User (returns either `True` or `False`). Defaults to all superusers.

GRAPPELLI_SWITCH_USER_TARGET A function which defines if a User is a valid switch target (returns either `True` or `False`). Defaults to all staff users, excluding superusers.

GRAPPELLI_CLEAN_INPUT_TYPES Replaces HTML5 input types (search, email, url, tel, number, range, date, month, week, time, datetime, datetime-local, color) due to browser inconsistencies. Set to `False` in order to not replace the mentioned input types.

1.2.2 Collapsibles

Use the `classes` property in order to define collapsibles for a ModelAdmin or an InlineModelAdmin. Possible values are `grp-collapse grp-open` and `grp-collapse grp-closed`:

```
class ModelOptions(admin.ModelAdmin):
    fieldsets = (
        ('', {
            'fields': ('title', 'subtitle', 'slug', 'pub_date', 'status',),
        }),
        ('Flags', {
            'classes': ('grp-collapse grp-closed',),
            'fields' : ('flag_front', 'flag_sticky', 'flag_allow_comments', 'flag_
↪comments_closed',),
        }),
        ('Tags', {
            'classes': ('grp-collapse grp-open',),
            'fields' : ('tags',),
        }),
    )

class StackedItemInline(admin.StackedInline):
    classes = ('grp-collapse grp-open',)

class TabularItemInline(admin.TabularInline):
    classes = ('grp-collapse grp-open',)
```

With StackedInlines, an additional property `inline_classes` is available to define the default collapsible state of inline items (as opposed to the inline group):

```
class StackedItemInline(admin.StackedInline):
    classes = ('grp-collapse grp-open',)
    inline_classes = ('grp-collapse grp-open',)
```

1.2.3 Inline Sortables

For using drag/drop with inlines, you need to add a `PositiveIntegerField` to your Model:

```
class MyInlineModel(models.Model):
    mymodel = models.ForeignKey(MyModel)
    # position field
    position = models.PositiveSmallIntegerField("Position", null=True)
    class Meta:
        ordering = ['position']
```

Now, define the `sortable_field_name` with your `InlineModelAdmin`:

```
class MyInlineModelOptions(admin.TabularInline):
    fields = (... , "position",)
    # define the sortable
    sortable_field_name = "position"
```

The inline rows are reordered based on the sortable field (with a templatetag `formsetsort`). When submitting a form, the values of the sortable field are reindexed according to the position of each row. We loop through each field of each row and check if the field has a value. If at least one value is given for a row, the sortable field is being updated. In order to exclude specific fields from this behaviour, use *Sortable Excludes*.

In case of errors (somewhere within the form), the position of inline rows is preserved. This also applies to rows prepared for deletion while empty rows are being moved to the end of the formset.

Besides using the drag/drop-handler, you are also able to manually update the position values. This is especially useful with lots of inlines. Just change the number within the position field and the row is automatically moved to the new position. Each row is being reindexed with submitting the form.

GrappelliSortableHiddenMixin

There is also `GrappelliSortableHiddenMixin`, which is a Mixin in order to hide the PositionField. Please note that this Mixin works with a default `sortable_field_name = "position"`. Therefore, you only need to explicitly define the `sortable_field_name` if it's named differently.

```
from grappelli.forms import GrappelliSortableHiddenMixin

class MyInlineModelOptions(GrappelliSortableHiddenMixin, admin.TabularInline):
    fields = (... , "position",)

# explicitly defining the sortable is only necessary
# if the sortable field name is not 'position'
class MyCustomInlineModelOptions(GrappelliSortableHiddenMixin, admin.TabularInline):
    fields = (... , "customposition",)
    sortable_field_name = "customposition"
```

Sortable Excludes

You may want to define `sortable_excludes` (either list or tuple) in order to exclude certain fields from having an effect on the position field. With the example below, the fields `field_1` and `field_2` have default values (so they are not empty with a new inline row). If we do not exclude this fields, the position field is updated for empty rows:

```
class MyInlineModelOptions(admin.TabularInline):
    fields = (... , "position",)
    # define the sortable
    sortable_field_name = "position"
    # define sortable_excludes
    sortable_excludes = ("field_1", "field_2",)
```

1.2.4 Rearrange Inlines

Sometimes it might make sense to not show inlines at the bottom of the page/form, but somewhere in–between. In order to achieve this, you need to define a placeholder with your fields/fieldsets in admin.py:

```
("Some Fieldset", {
    "classes": ("grp-collapse grp-open",),
    "fields": ("whatever",)
}),
("Image Inlines", {"classes": ("placeholder images-group",), "fields" : ()}),
("Another Fieldset", {
    "classes": ("grp-collapse grp-open",),
    "fields": ("whatever",)
}),

inlines = [ImageInlines]
```

The two classes for the placeholder are important. First, you need a class `placeholder`. The second class has to match the `id` of the inline–group.

1.2.5 Related Lookups

With Grappelli, you're able to add the representation of an object beneath the input field (for fk– and m2m–fields):

```
class MyModel(models.Model):
    related_fk = models.ForeignKey(RelatedModel, verbose_name=u"Related Lookup (FK)")
    related_m2m = models.ManyToManyField(RelatedModel, verbose_name=u"Related Lookup␣
↪(M2M)")

class MyModelOptions(admin.ModelAdmin):
    # define the raw_id_fields
    raw_id_fields = ('related_fk','related_m2m',)
    # define the related_lookup_fields
    related_lookup_fields = {
        'fk': ['related_fk'],
        'm2m': ['related_m2m'],
    }
```

With generic relations, related lookups are defined like this:

```
from django.contrib.contenttypes import generic
from django.contrib.contenttypes.models import ContentType
from django.db import models

class MyModel(models.Model):
    # first generic relation
    content_type = models.ForeignKey(ContentType, blank=True, null=True, related_name=
↪"content_type")
    object_id = models.PositiveIntegerField(blank=True, null=True)
    content_object = generic.GenericForeignKey("content_type", "object_id")
    # second generic relation
    relation_type = models.ForeignKey(ContentType, blank=True, null=True, related_
↪name="relation_type")
    relation_id = models.PositiveIntegerField(blank=True, null=True)
    relation_object = generic.GenericForeignKey("relation_type", "relation_id")

class MyModelOptions(admin.ModelAdmin):
    # define the related_lookup_fields
    related_lookup_fields = {
        'generic': [['content_type', 'object_id'], ['relation_type', 'relation_id']],
    }
```

If your generic relation points to a model using a custom primary key, you need to add a property `id`:

```
class RelationModel(models.Model):
    cpk  = models.IntegerField(primary_key=True, unique=True, editable=False)

    @property
    def id(self):
        return self.cpk
```

For the representation of an object, we first check for a callable `related_label`. If not given, `__unicode__` is being used in Python 2.x or `__str__` in Python 3.x.

Example in Python 2:

```
def __unicode__(self):
    return u"%s" % self.name

def related_label(self):
    return u"%s (%s)" % (self.name, self.id)
```

Example in Python 3:

```
def __str__(self):
    return "%s" % self.name

def related_label(self):
    return "%s (%s)" % (self.name, self.id)
```

Note: In order to use related lookups, you need to register both ends (models) of the relationship with your `admin.site`.

1.2.6 Autocomplete Lookups

Autocomplete lookups are an alternative to related lookups (for foreign keys, many–to-many relations and generic relations).

Add the staticmethod `autocomplete_search_fields` to all models you want to search for:

```
class MyModel(models.Model):
    name = models.CharField(u"Name", max_length=50)

    @staticmethod
    def autocomplete_search_fields():
        return ("id__iexact", "name__icontains",)
```

If the staticmethod is not given, `GRAPPELLI_AUTOCOMPLETE_SEARCH_FIELDS` will be used if the app/model is defined:

```
GRAPPELLI_AUTOCOMPLETE_SEARCH_FIELDS = {
    "myapp": {
        "mymodel": ("id__iexact", "name__icontains",)
    }
}
```

Defining autocomplete lookups is very similar to related lookups:

```
class MyModel(models.Model):
    related_fk = models.ForeignKey(RelatedModel, verbose_name=u"Related Lookup (FK)")
    related_m2m = models.ManyToManyField(RelatedModel, verbose_name=u"Related Lookup␣
↪(M2M)")

class MyModelOptions(admin.ModelAdmin):
    # define the raw_id_fields
    raw_id_fields = ('related_fk','related_m2m',)
    # define the autocomplete_lookup_fields
    autocomplete_lookup_fields = {
        'fk': ['related_fk'],
```

(continues on next page)

(continued from previous page)

```
        'm2m': ['related_m2m'],
    }
```

This also works with generic relations:

```
from django.contrib.contenttypes import generic
from django.contrib.contenttypes.models import ContentType
from django.db import models

class MyModel(models.Model):
    # first generic relation
    content_type = models.ForeignKey(ContentType, blank=True, null=True, related_name=
↪"content_type")
    object_id = models.PositiveIntegerField(blank=True, null=True)
    content_object = generic.GenericForeignKey("content_type", "object_id")
    # second generic relation
    relation_type = models.ForeignKey(ContentType, blank=True, null=True, related_
↪name="relation_type")
    relation_id = models.PositiveIntegerField(blank=True, null=True)
    relation_object = generic.GenericForeignKey("relation_type", "relation_id")

class MyModelOptions(admin.ModelAdmin):
    # define the autocomplete_lookup_fields
    autocomplete_lookup_fields = {
        'generic': [['content_type', 'object_id'], ['relation_type', 'relation_id']],
    }
```

If your generic relation points to a model using a custom primary key, you need to add a property `id`:

```
class RelationModel(models.Model):
    cpk  = models.IntegerField(primary_key=True, unique=True, editable=False)

    @property
    def id(self):
        return self.cpk
```

If the human-readable value of a field you are searching on is too large to be indexed (e.g. long text as SHA key) or is saved in a different format (e.g. date as integer timestamp), add a staticmethod `autocomplete_term_adjust` to the corresponding model with the appropriate transformation and perform the lookup on the indexed field:

```
class MyModel(models.Model):
    text = models.TextField(u"Long text")
    text_hash = models.CharField(u"Text hash", max_length=40, unique=True)

    @staticmethod
    def autocomplete_term_adjust(term):
        return hashlib.sha1(term).hexdigest()

    @staticmethod
    def autocomplete_search_fields():
        return ("text_hash__iexact",)
```

For the representation of an object, we first check for a callable `related_label`. If not given, `__unicode__` is being usedin Python 2.x or `__str__` in Python 3.x.

Example in Python 2:

```
def __unicode__(self):
    return u"%s" % self.name

def related_label(self):
    return u"%s (%s)" % (self.name, self.id)
```

Example in Python 3:

```
def __str__(self):
    return "%s" % self.name

def related_label(self):
    return "%s (%s)" % (self.name, self.id)
```

Note: In order to use autocompletes, you need to register both ends (models) of the relationship with your `admin.site`.

1.2.7 Using TinyMCE

Grappelli already comes with TinyMCE and a minimal theme as well. In order to use TinyMCE, copy `tinymce_setup.js` to your static directory, adjust the setup (see TinyMCE Configuration) and add the necessary javascripts to your ModelAdmin definition (see ModelAdmin asset definitions):

```
class Media:
    js = [
        '/static/grappelli/tinymce/jscripts/tiny_mce/tiny_mce.js',
        '/static/path/to/your/tinymce_setup.js',
    ]
```

Using TinyMCE with inlines is a bit more tricky because of the hidden extra inline. You need to write a custom template and use the inline callbacks to

- `onInit`: remove TinyMCE instances from the empty form.
- `onAfterAdded`: initialize TinyMCE instance(s) from the form.
- `onBeforeRemoved`: remove TinyMCE instance(s) from the form.

Note: TinyMCE with inlines is not supported by default.

If our version of TinyMCE does not fit your needs, add a different version to your static directory and change the above mentioned ModelAdmin setup (paths to js–files).

Warning: TinyMCE will be removed with version 3.0 of Grappelli, because TinyMCE version 4.x comes with a decent skin.

1.2.8 Changelist Templates

Grappelli comes with different change–list templates. To use the alternative templates, you need to add `change_list_template` to your ModelAdmin definition.

Filters as drop-down, automatically applied (default template)

The default template shows filters as a drop–down, selecting a single filter immediately applies the filter. This template supports use cases where single filters should be quickly applicable.

```
class MyModelOptions(admin.ModelAdmin):
    change_list_template = "admin/change_list.html"
```

Filters as drop-down, manually applied

This template shows filters as a drop-down, selected filters have to be applied manually by clicking an "Apply" button. This template supports use cases where multiple filters have to be selected and applied at the same time.

```
class MyModelOptions(admin.ModelAdmin):
    change_list_template = "admin/change_list_filter_confirm.html"
```

Filters in a sidebar, automatically applied

This template shows filters in a sidebar, selecting a single filter immediately applies the filter. This template supports use cases where single filters should be quickly applicable.

```
class MyModelOptions(admin.ModelAdmin):
    change_list_template = "admin/change_list_filter_sidebar.html"
```

Filters in a sidebar, manually applied

This template shows filters in a sidebar, selected filters have to be applied manually by clicking an "Apply" button. This template supports use cases where multiple filters have to be selected and applied at the same time.

```
class MyModelOptions(admin.ModelAdmin):
    change_list_template = "admin/change_list_filter_confirm_sidebar.html"
```

1.2.9 Changelist Filters

Grappelli comes with 2 different change–list filters. The standard filters are selects, the alternative filters are list of options (similar to djangos admin interface). To use the alternative filters, you need to add `change_list_filter_template` to your ModelAdmin definition:

```
class MyModelOptions(admin.ModelAdmin):
    change_list_filter_template = "admin/filter_listing.html"
```

1.2.10 Switch User

You sometimes might need to see the admin interface as a different user (e.g. in order to verify if permissions are set correctly or to follow an editors explanation). If you set `GRAPPELLI_SWITCH_USER` to `True`, you'll get additional users with your user dropdown. Moreover, you can easily switch back to the original User.

Note: This functionality might change with future releases.

Warning: If you are using a custom user model and want to turn this feature on, pay attention to the following topics:

- if `is_superuser` is neither a field nor a property of your user model, you will have to set both `GRAPPELLI_SWITCH_USER_ORIGINAL` and `GRAPPELLI_SWITCH_USER_TARGET` to functions; failing to do so will break the admin area. If you followed the instructions in the Django docs, `is_superuser` won't be a field nor a property of your user model. If you define `is_superuser` as a property of your model, the admin area will get back to work.
- if `is_staff` is not a field, and/or `is_superuser` is neither a field nor a property of your user model, the Grappelli tests will be broken (because e.g. of some `user.is_staff = True` instructions). This - again- is your case if you followed the Django docs on customizing user model, where `is_staff` is defined as a property (as opposite to a field).

1.2.11 Clean input types

With setting `GRAPPELLI_CLEAN_INPUT_TYPES` to `True`, Grappelli automatically replaces all HTML5 input types (search, email, url, tel, number, range, date month, week, time, datetime, datetime-local, color) with `type="text"`. This is useful if you want to avoid browser inconsistencies with the admin interface. Moreover, you remove frontend form validation and thereby ensure a consistent user experience.

Note: This functionality might change with future releases.

CHAPTER 2

Dashboard

2.1 Dashboard Setup

With the Django admin interface, the admin index page reflects the structure of your applications/models. With `grappelli.dashboard` you are able to change that structure and rearrange (or group) apps and models.

Note: `grappelli.dashboard` is a simplified version of Django Admin Tools: Bookmarks, Menus and the custom App Index are **not available with Grappelli**.

Open `settings.py` and add `grappelli.dashboard` to your `INSTALLED_APPS` (before `grappelli`, but after `django.contrib.contenttypes`). Check if the request context processor is being used:

```
INSTALLED_APPS = (
    'django.contrib.contenttypes',
    'grappelli.dashboard',
    'grappelli',
    'django.contrib.admin',
)

TEMPLATES = [
    {
        ...
        'OPTIONS': {
            'context_processors': [
                ...
                'django.template.context_processors.request',
                ...
            ],
        },
    },
]
```

2.1.1 Custom dashboard

To customize the index dashboard, you first need to add a custom dashboard, located within your project directory. Depending on the location of manage.py, you might need to add the project directory to the management command (see last example below):

```
$ python manage.py customdashboard  # creates dashboard.py
$ python manage.py customdashboard somefile.py  # creates somefile.py
$ python manage.py customdashboard projdir/somefile.py  # creates somefile.py in␣
↪projdir
```

The created file contains the class `CustomIndexDashboard` that corresponds to the admin index page dashboard. Now you need to add your custom dashboard. Open your `settings.py` file and define `GRAPPELLI_INDEX_DASHBOARD`:

```
GRAPPELLI_INDEX_DASHBOARD = 'yourproject.dashboard.CustomIndexDashboard'
GRAPPELLI_INDEX_DASHBOARD = {  # alternative method
    'yourproject.admin.admin_site': 'yourproject.my_dashboard.CustomIndexDashboard',
}
```

If you're using a custom admin site (not `django.contrib.admin.site`), you need to define the dashboard with the alternative method.

2.1.2 Custom dashboards for multiple sites

If you have several admin sites, you need to create a custom dashboard for each site:

```
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)),
)
```

To configure your dashboards, you could do:

```
$ python manage.py customdashboard dashboard.py
$ python manage.py customdashboard my_dashboard.py
```

Open your `settings.py` file and define `GRAPPELLI_INDEX_DASHBOARD`:

```
GRAPPELLI_INDEX_DASHBOARD = {
    'django.contrib.admin.site': 'yourproject.dashboard.CustomIndexDashboard',
    'yourproject.admin.admin_site': 'yourproject.my_dashboard.CustomIndexDashboard',
}
```

2.2 Dashboard API

This section describe the API of the Grappelli dashboard and dashboard modules.

2.2.1 The Dashboard class

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

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

template The template used to render the dashboard. Default: `grappelli/dashboard/dashboard.html`

Here's an example of a custom dashboard:

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

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

        # append an app list module for "Applications"
        self.children.append(modules.AppList(
            title=_('Applications'),
            column=1,
            collapsible=True,
            exclude=('django.contrib.*',),
        ))

        # append an app list module for "Administration"
        self.children.append(modules.AppList(
            title=_('Administration'),
            column=1,
            collapsible=True,
            models=('django.contrib.*',),
        ))

        # append a recent actions module
        self.children.append(modules.RecentActions(
            title=_('Recent Actions'),
            column=2,
            collapsible=False,
            limit=5,
        ))
```

2.2.2 The DashboardModule class

Base class for all dashboard modules. Dashboard modules have the following properties:

collapsible Boolean that determines whether the module is collapsible. Default: `True`

column (required) Integer that corresponds to the column. Default: `None`

title String that contains the module title, make sure you use the django gettext functions if your application is multilingual. Set to `''` if you need to suppress the title.

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

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

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

template The template used to render the module. Default: `grappelli/dashboard/module.html`

2.2.3 The Group class

Represents a group of modules:

```
from grappelli.dashboard import modules, Dashboard

class MyDashboard(Dashboard):
    def __init__(self, **kwargs):
        Dashboard.__init__(self, **kwargs)
        self.children.append(modules.Group(
            title="My group",
            column=1,
            collapsible=True,
            children=[
                modules.AppList(
                    title='Administration',
                    models=('django.contrib.*',)
                ),
                modules.AppList(
                    title='Applications',
                    exclude=('django.contrib.*',)
                )
            ]
        ))
```

2.2.4 The LinkList class

A module that displays a list of links.

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.

`target` A string or boolean value describing what is the link target. To open link in a new window/tab you can pass `True` or `'_blank'` value to this parameter. When you pass an string value, it is directly used in the `target` attribute of the generated `a` tag in the template.

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

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

```
from grappelli.dashboard import modules, Dashboard

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

        self.children.append(modules.LinkList(
            title='Links',
            column=2,
            children=(
                {
                    'title': 'Python website',
```

(continues on next page)

(continued from previous page)

```
                    'url': 'http://www.python.org',
                    'external': True,
                    'description': 'Python programming language rocks!',
                    'target': '_blank',
                },
                ['Django website', 'http://www.djangoproject.com', True],
                ['Some internal link', '/some/internal/link/'],
            )
        ))
```

2.2.5 The AppList class

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.models.BlogEntry") 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.models.BlogEntry") match an element of this list (e.g. "blog.*") it won't appear in the dashboard module.

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

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

```
from grappelli.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',
            column=1,
            models=('django.contrib.*',)
        ))
        # will list all apps except the django.contrib ones
        self.children.append(modules.AppList(
            title='Applications',
            column=1,
            exclude=('django.contrib.*',)
        ))
```

Note: 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 won't be displayed.

2.2.6 The ModelList class

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.models.BlogEntry") 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.models.BlogEntry") match an element of this list (e.g. "blog.*") it won't appear in the dashboard module.

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

```
from grappelli.dashboard import modules, Dashboard

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

        self.children.append(modules.ModelList(
            title='Several Models',
            column=1,
            models=('django.contrib.*',)
        ))

        self.children.append(modules.ModelList(
            title='Single Model',
            column=1,
            models=('blog.models.BlogEntry',)
        ))
```

Note: 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 won't be displayed.

2.2.7 The RecentActions class

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: `10`

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

```
from grappelli.dashboard import modules, Dashboard

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

        self.children.append(modules.RecentActions(
            title='Django CMS recent actions',
            column=3,
            limit=5,
        ))
```

2.2.8 The Feed class

Class that represents a feed dashboard module.

Note: This class requires the Universal Feed Parser module, so you'll need to install it.

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: `None` (which means that all children are displayed)

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

```
from grappelli.dashboard import modules, Dashboard

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

        self.children.append(modules.Feed(
            title=_('Latest Django News'),
            feed_url='http://www.djangoproject.com/rss/weblog/',
            column=3,
            limit=5,
        ))
```

CHAPTER 3

Internals

3.1 Internals

3.1.1 Templates

Grappelli includes a Documentation about the HTML/CSS framework. If you're using the default URL-pattern (see *Quick start guide*) you'll find the documentation at `/grappelli/grp-doc/` (by default, these URLs are commented out).

3.1.2 Javascripts

Grappelli only uses a subset of the original admin javascripts.

- If there's only minor modifications, we use the original javascript (e.g. `RelatedObjectLookups.js`).
- If we add functionality, we use our own jQuery-plugins (e.g. `jquery.grp_collapsible.js`) or jQuery-widgets (e.g. `jquery.grp_timepicker.js`)
- If there are major modifications, we use our own files as well (e.g. `jquery.grp_inlines.js`).

Original Javascripts

in `/static/admin/js/`

`calendar.js`, `collapse.js`, `collapse.min.js`, `inlines.js`, `inlines.min.js`, `jquery.init.js`, `jquery.js`, `j`
not used (empty to prevent 404)

`actions.js`, `actions.min.js` minor modifiations, marked with GRAPPELLI CUSTOM

`core.js` original

`prepopulate`, `prepopulate.min.js` original

`related-widget-wrapper.js` original

SelectBox.js original

SelectFilter2.js minor modifiations, marked with GRAPPELLI CUSTOM (e.g. removed help-text, because trusted editors should know what to do)

urlify.js original

RelatedObjectLookups.js minor modifiations, marked with GRAPPELLI CUSTOM

Grappelli Javascripts

in `/static/admin/js/`

jquery.grp_collapsible.js collapsibles

jquery.grp_collapsible_group.js grouped collapsibles (inlines)

jquery.grp_inlines.js inlines (tabular and stacked)

jquery.grp_related_fk.js foreign-key lookup

jquery.grp_related_m2m.js m2m lookup

jquery.grp_related_generic.js generic lookup

jquery.grp_autocomplete_fk.js foreign-key lookup with autocomplete

jquery.grp_autocomplete_m2m.js m2m lookup with autocomplete

jquery.grp_autocomplete_generic.js generic lookup with autocomplete

jquery.grp_timepicker.js timepicker

grappelli.js main grappelli js

grappelli.min.js minified version of all Grappelli JS

CHAPTER 4

Help

4.1 Help

4.1.1 FAQ

Some questions, some answers.

Why should I use Grappelli?

If you are pleased with how the original admin interface looks, you probably shouldn't. Grappelli is mainly about a consistent, grid-based style.

I need help!

see *Troubleshooting*.

Which Browser do I need with Grappelli?

We are testing with Firefox, Chrome and Safari. IE9 and Opera should work fine as well.

Can I use another editor than TinyMCE?

Of course (better use markdown anyway).

Why don't you use Twitter Bootstrap?

For now, custom Compass-based stylesheets gives us much more options. But we do understand the need to easily customize the admin interface and we're discussing this issue on a regular basis.

Do you guys cooperate with the Django–Devs?

Occasionally, we have been discussing features and implementations in the past.

How can I contribute?

Help is very much appreciated.

- Fork django-grappelli and submit feedback/patches.
- Work on Django Tickets related to contrib.admin.
- Take a look at *Django Issues*.

What is your goal with Grappelli?

Our goal is that Grappelli will be redundant eventually (though we doubt that will happen).

Who develops Grappelli?

Grappelli is developed and maintained by Patrick Kranzlmüller & Axel Swoboda of vonautomatisch.

4.1.2 Best Practice

Some tips in order to make the most of Grappelli.

M2M horizontal/vertical

Most of the time, it's probably better to use Autocompletes.

Radiolists/Checkboxlists

If you're having more than (say) 5 items to choose from, avoid using this (esp. with tabular/stacked Inlines).

4.1.3 Third Party Applications

A list of 3rd-party applications compatible with Grappelli. Pleaes note that except of Django FileBrowser, these apps are not maintained by the team behind Grappelli. Compatibility with these apps is therefore not guaranteed and heavily relies on patches and pull-requests by the community.

Django FileBrowser

No additional setup is needed when installing the Django FileBrowser with Grappelli.

Django Reversion

Grappelli includes all necessary templates for Django Reversion.

Note: `grappelli` needs to be before `reversion` within `INSTALLED_APPS`.

Django Smuggler

Grappelli includes all necessary templates for Django Smuggler.

Note: `grappelli` needs to be before `smuggler` within `INSTALLED_APPS`.

Django Constance

Grappelli includes the necessary template for Django Constance.

Note: `grappelli` needs to be before `constance` within `INSTALLED_APPS`.

Django Import-Export

Grappelli includes the necessary template for Django Import-Export.

Note: `grappelli` needs to be before `import_export` within `INSTALLED_APPS`.

4.1.4 Troubleshooting

Sometimes you might have a problem installing/using Grappelli.

Check your setup

First, please check if the problem is caused by your setup.

- Read *Quick start guide* and *Customization*.
- Check if the static/media files are served correctly.
- Make sure you have removed all customized admin templates from all locations in `TEMPLATE_DIRS` paths or check that these templates are compatible with Grappelli.

Check issues

If your setup is fine, please check if your problem is a known issue.

- Read *Django Issues* in order to see if your problem is indeed related to Grappelli.
- Take a look at all Grappelli Issues (incuding closed) and search the Grappelli Google-Group.

Add a ticket

If you think you've found a bug, please add a ticket and follow the guidelines for contributing.

4.1.5 Django Issues

There are some known problems with the Django admin interface. I'm going to list them here in order to avoid confusion (because the problems are not related to Grappelli whatsoever).

see http://code.djangoproject.com/wiki/DjangoDesign

Harcoded Stuff

This means HTML markup within views (instead of using templates). There's a lot of this within the admin interface and therefore it's just not possible to style some elements. For other elements, we need to use ugly hacks or strange CSS.

The solution is to implement floppy-forms (https://github.com/brutasse/django-floppyforms) with Django.

Javascripts

Some Javascripts are about 5 years old. Others are pretty new. Some are jQuery, some not. Still a bit messy and hard to customize.

see *Javascripts*.

Reordering Edit-Inlines

First, the `can_order` attribute is not available with the admin interface. Second, in case of errors, formsets are not returned in the right order. Therefore, reordering inlines is currently only possible with some hacks.

see http://code.djangoproject.com/ticket/14238

HTML5 input types

There is unpredictable behaviour with certain input types (e.g. number) and Django should remove these types from our point of view (at least with the admin interface). Moreover, form validation should not be moved to the frontend of the admin interface.

see https://code.djangoproject.com/ticket/23075 (although this ticket is marked with "fixed", the behaviour is still inconsistent)

The Admin Index Site

Currently, the admin index site reflects the structure of your applications/models. We don't think editors (who use the admin site) are interested in the structure of your project/applications. What they want is the most reasonable list of models, divided into different sections (not necessarily apps).

see http://code.djangoproject.com/ticket/7497

The App Index

Again, we don't think customers/editors are interested in your apps.

Related Lookups

With either `raw_id_fields` or `Generic Relations`, the representation for an object should be displayed beneath the input-field. When changing the `object-id` (or selecting an object with the related pop-up window) the representation should be updated.

This issue is solved with Grappelli (unfortunately overly complex due to the limitations of the original admin interface).

Autocompletes

As an alternative to `Related Lookups` it should also be possible to implement `Autocompletes`. Grappelli includes `Autocompletes`, but it should be possible without hacking the admin-interface.

Help text and Many-to-Many Fields

The `help_text` doesn't show up with M2M-Fields, when using the RawID-Widget (e.g. with Autocompletes). Nothing we can do about that.

Searching Generic Relations

It's not possible to use a `content_object` within search_fields.

Javascript loading

Unfortunately, it's not possible to combine all django javascripts.

Admin Documentation

The document structure of the admin_doc templates is messy (about every second template has a different structure). Therefore, it's hard to style these pages. Trying to do our best to give it a decent look though.

HTML/CSS Framework

For the admin interface to be customizable, flexible and extensible, we need a coherent HTML/CSS scheme.

We do think that Grappelli is a first step.

4.1.6 Grappelli 2.13.x Release Notes

Grappelli 2.13.x is compatible with Django 2.2.

Update from Grappelli 2.12.x

- Update Django to 2.2 and check https://docs.djangoproject.com/en/dev/releases/2.2/
- Update Grappelli to 2.13.x

4.2 Contributing

We are happy to see patches and improvements with Grappelli. But please keep in mind that there are some guidelines you should follow.

To file an issue with Grapelli, see contributing on github.

4.2.1 Requirements

For working with Javascript and CSS, you need Node, Ruby, Grunt, Sass and Compass. In order to update the documentation, Sphinx and the Sphinx RTD Theme have to be installed. Finally, you should install flake8 when working with python files.

It's out of the scope of this tutorial to go into details, but you should find lots of useful references on how to install these dependencies.

Node is needed for Grunt, Ruby for Sass/Compass:

```
brew install node
brew install ruby
```

Now you are able to install Grunt and Compass (Sass is automatically installed with Compass):

```
npm install -g grunt-cli
gem install compass
```

Change to the root of your grappelli installation, where `package.json` and `Gruntfile.js` are located and install the Grunt dependencies:

```
npm install
```

Start your virtual environment and install the python dependencies:

```
pip install sphinx
pip install sphinx-rtd-theme
pip install flake8
```

4.2.2 Branches

Please commit to the stable branch of a specific Grappelli version and do not use the master branch. For example, in order to send pull-requests for Grappelli 2.7, use the branch stable/2.7.x.

4.2.3 Python

When working with python files, please refer to the Django Coding Guidelines. Grappelli includes a grunt task which checks for coding errors (you should always use this task if you update .py files):

```
grunt flake8
```

Note: flake8 has to be installed in order for this task to work.

4.2.4 Javascripts & Stylesheets

If you change any of the Grappelli javascripts, you need to jshint the files and create grappelli.min.js:

```
grunt javascripts
```

When working with CSS (which is .scss in our case), you have to compile with:

```
grunt compass
```

4.2.5 Documentation

If you update documentation files, there's a grunt task for building the html files (this is not needed with a pull-request, but you might wanna check your updates locally):

```
grunt sphinx
```

4.2.6 Watch

You can use `grunt watch` or just `grunt` in order to check for live update on js/scss files as well as the documentation and run the necessary grunt tasks in the background while working.

4.3 Changelog

4.3.1 2.13.4 (not yet released)

4.3.2 2.13.3 (December 27th 2019)

- Fixed: Horizontal scrolling.
- Fixed: Changelist with custom filters.
- Fixed: Form select icons (Chrome).

4.3.3 2.13.2 (November 11th 2019)

- Fixed: added Django autocomplete JS files.
- Fixed: added minified jQuery 3.3.1 file.
- Fixed: added request to *formfield_for_dbfield*.
- Fixed: use safe label with autocompletes.
- Improved: added separate file for documentation URLs.

- Improved: removed Python 2 support.
- Improved: CSS footer and submit-row fixes.
- Improved: CSS field width with inline tabular.

4.3.4 2.13.1 (June 25th 2019)

- First release of Grappelli which is compatible with Django 2.2.

CHAPTER 5

Code

https://github.com/sehmaschine/django-grappelli

CHAPTER 6

Website

http://www.grappelliproject.com

CHAPTER 7

Discussion

Use the Grappelli Google Group to ask questions or discuss features.

CHAPTER 8

Versions and Compatibility

Grappelli is always developed against the latest stable Django release and is NOT tested with Django's master branch.

- Grappelli 2.13.3 (December 27th, 2019): Compatible with Django 2.2
- Grappelli 2.12.4 (November 11th, 2019): Compatible with Django 2.1
- Grappelli 2.10.4 (November 1st, 2018): Compatible with Django 1.11 (LTS)

Current development branches:

- Grappelli 2.13.4 (Development version for Django 2.2, see branch Stable/2.13.x)
- Grappelli 2.12.5 (Development version for Django 2.1, see branch Stable/2.12.x)
- Grappelli 2.10.5 (Development version for Django 1.11, see branch Stable/2.10.x)

Older versions are available at GitHub, but are not supported anymore.