Mastering Django Admin

ChillarAnand

Sep 17, 2019

CONTENTS

CHAPTER

ONE

PREFACE

1.1 Why this book?

blog posts

1.2 Who should read this book?

users

1.3 Acknowlodgements

Krace Kumar

Haris Ibrahim

Tim Graham,https://techytim.com/

Andrew Godwin, http://www.aeracode.org/

Haki Banita

https://hakibenita.com/

CHAPTER

TWO

THE MILLION DOLLAR ADMIN

not to use for external applications

Not good ux

Django admin was first released in 2005 and it has gone through a lot of changes since then.

mental models

Revamp

reduce cost of maintainance and development

people come to django becase of admin

more visual and responsive

https://github.com/sshwsfc/xadmin

https://jacobian.org/2016/may/26/so-you-want-a-new-admin/

CHAPTER

THREE

BETTER DEFAULTS

3.1 Use ModelAdmin

When a model is registered with admin, it just shows the string representation of the model object in changelist page.

```
from book.models import Book

admin.site.register(Book)
```

Django provides ModelAdmin[1] class which represents a model in admin. We can use the following options to make the admin interface informative and easy to use.

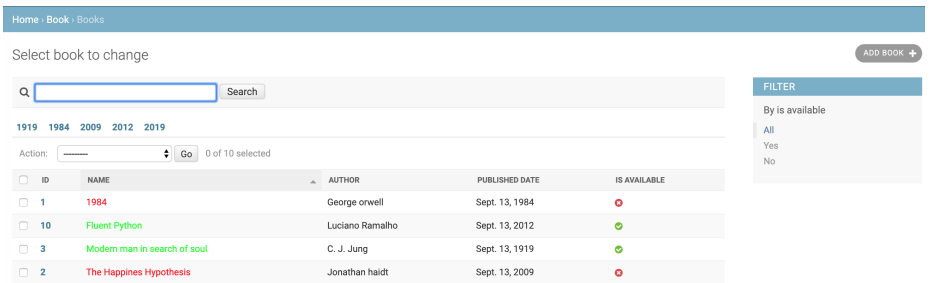
- *list_display* to display required fields and add custom fields.
- *list_filter* to add filters data based on a column value.

[1] https://docs.djangoproject.com/en/2.2/ref/contrib/admin/#modeladmin-objects

- *list_per_page* to set how many items to be shown on paginated page.
- *search_fields* to search for records based on a field value.
- *date_hierarchy* to provide date-based drilldown navigation for a field.
- *readonly_fields* to make seleted fields readonly in edit view.
- *prepopulated_fields* to auto generate a value for a column based on another column.
- *save_as* to enable save as new in admin change forms.

```
from book.models import Book
from django.contrib import admin


@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
    list_display = ('id', 'name_colored', 'author',
↪'published_date', 'cover', 'is_available')
    list_filter = ('is_available',)
    list_per_page = 10
    search_fields = ('name',)
    date_hierarchy = 'published_date'
    readonly_fields = ('created_at', 'updated_at')

    def name_colored(self, obj):
        if obj.is_available:
            color_code = '00FF00'
        else:
            color_code = 'FF0000'
        html = '<span style="color: #{};">{}</span>
↪'.format(color_code, obj.name)
        return format_html(html)

    name_colored.admin_order_field = 'name'
    name_colored.short_description = 'name'
```

In *list_display* in addition to columns, we can add custom methods which can be used to show calculated fields. For example, we can change book color based on its availability.



3.2 Use better widgets

Sometimes widgets provided by Django are not handy to the users. In such cases it is better to add tailored widgets based on the field.

For images, instead of showing a link, we can show thumbnails of images so that users can see the picture in the list view itself.

```
@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
  list_display = ('id', 'name_colored', 'author',
↪'published_date', 'is_available')

  def thumbnail(self, obj):
    width, height = 200, 400
    html = '<img src="/{url}" width="{width}"␣
↪height={height} />'
    return format_html(
        html.format(url=obj.cover.url, width=width,␣
↪height=height)
    )
```

Viewing and editing JSON field in admin interface will be very difficult in the textbox. Instead, we can use JSON Editor widget provided any

third-party packages like django-json-widget, with which viewing and editing JSON data becomes much intuitive.

```
from django.contrib.postgres import fields
from django_json_widget.widgets import␣
↪JSONEditorWidget

@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
    formfield_overrides = {
        fields.JSONField: {
            'widget': JSONEditorWidget
        },
    }
```

When a choice field has few options, instead of showing dropdown, show radio buttons to make easy to choose.

wsywig editor for rich text.

3.3 Sorting Models By Frequency

https://github.com/mishbahr/django-modeladmin-reorder

3.4 Customize Header/Title

```
admin.site.site_header = 'My administration'
```

3.5 Provide better defaults for model fields

```
class Category(models.Model):
    name = ''
```

(continues on next page)

(continued from previous page)

```
    class Meta:
        verbose_name_plural = "categories"
```

Plural name

CHAPTER

FOUR

MANAGING MODEL RELATIONSHIPS

https://djangosnippets.org/snippets/2217/

4.1 Auto completion

- *autocomplete_fields*

    autocomplete_fields = [‘author’]

for foreignkey fields

4.2 Hyperlink Foreignkeys To Its Change View In Admin

Consider Book model which has Author as foreignkey.

```
from django.db import models


class Author(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
```

(continues on next page)

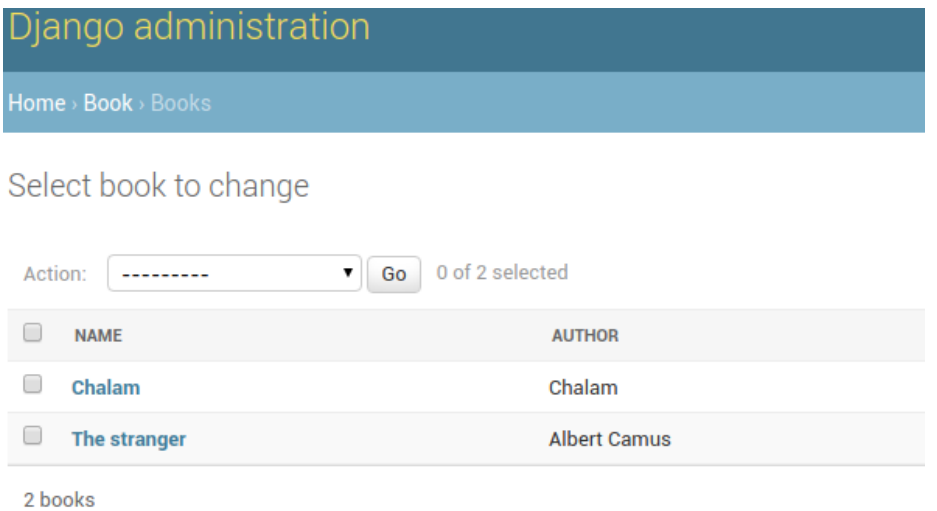
(continued from previous page)

```
    title = models.CharField(max_length=100)
    author = models.ForeignKey(Author)
```

We can register these models with admin interface as follows.

```
from django.contrib import admin

from .models import Author, Book

class BookAdmin(admin.ModelAdmin):
    list_display = ('name', 'author', )

admin.site.register(Author)
admin.site.register(Book, BookAdmin)
```

Once they are registered, admin page shows Book model like this.



While browsing books, we can see book name and author name. Here, book name field is liked to book change view. But author field is shown as plain text.

If we have to modify author name, we have to go back to authors admin page, search for relevant author and then change name.

This becomes tedious if users spend lot of time in admin for tasks like this. Instead, if author field is hyperlinked to author change view, we can directly go to that page and change the name.

Django provides an option to access admin views by its URL reversing system. For example, we can get change view of author model in book app using reverse("admin:book_author_change", args=id). Now we can use this url to hyperlink author field in book admin.

```
from django.contrib import admin
from django.utils.safestring import mark_safe


class BookAdmin(admin.ModelAdmin):
    list_display = ('name', 'author_link', )

    def author_link(self, book):
        url = reverse("admin:book_author_change",␣
↪args=[book.author.id])
        link = '<a href="%s">%s</a>' % (url, book.
↪author.name)
        return mark_safe(link)
    author_link.short_description = 'Author'
```

Now in the book admin view, author field will be hyperlinked to its change view and we can visit just by clicking it.

Depending on requirements, we can link any field in django to other fields or add custom fields to improve productivity of users in admin.

Custom hyper links

https://docs.djangoproject.com/en/dev/ref/models/instances/#get-absolute-url

4.3 Allow ForeignKey Fields In Admin List Display

Django admin has *ModelAdmin* class which provides options and functionality for the models in admin interface. It has options like *list_display*, *list_filter*, *search_fields* to specify fields for corresponding actions.

search_fields, *list_filter* and other options allow to include a ForeignKey or ManyToMany field with lookup API follow notation. For example, to search by book name in Bestselleradmin, we can specify *book__name* in search fields.

```
from django.contrib import admin

from book.models import BestSeller


class BestSellerAdmin(RelatedFieldAdmin):
    search_fields = ('book__name', )
    list_display = ('id', 'year', 'rank', 'book')


admin.site.register(Bestseller, BestsellerAdmin)
```

However Django doesn't allow the same follow notation in *list_display*. To include ForeignKey field or ManyToMany field in the list display, we have to write a custom method and add this method in list display.

```
from django.contrib import admin

from book.models import BestSeller


class BestSellerAdmin(RelatedFieldAdmin):
    list_display = ('id', 'rank', 'year', 'book',
↪'author')
```

(continues on next page)

(continued from previous page)

```
    search_fields = ('book__name', )

    def author(self, obj):
        return obj.book.author
    author.description = 'Author'


admin.site.register(Bestseller, BestsellerAdmin)
```

This way of adding foreignkeys in list_display becomes tedious when there are lots of models with foreignkey fields.

We can write a custom admin class to dynamically set the methods as attributes so that we can use the ForeignKey fields in list_display.

```
def get_related_field(name, admin_order_field=None,␣
↪short_description=None):
    related_names = name.split('__')

    def dynamic_attribute(obj):
        for related_name in related_names:
            obj = getattr(obj, related_name)
            return obj

    dynamic_attribute.admin_order_field = admin_
↪order_field or name
    dynamic_attribute.short_description = short_
↪description or related_names[-1].title().replace(
↪'_', ' ')
    return dynamic_attribute


class RelatedFieldAdmin(admin.ModelAdmin):
    def __getattr__(self, attr):
        if '__' in attr:
            return get_related_field(attr)
```

(continues on next page)

(continued from previous page)

```
        # not dynamic lookup, default behaviour
        return self.__getattribute__(attr)


class BestSellerAdmin(RelatedFieldAdmin):
    list_display = ('id', 'rank', 'year', 'book',
↪'book__author')
```

By sublcassing RelatedFieldAdmin, we can directly use foreignkey fields in list display.

However, this will lead to N+1 problem. We will discuss more about this and how to fix this in orm optimizations chapter.

https://github.com/theatlantic/django-nested-admin

CHAPTER

FIVE

AUTO REGISTER ALL MODELS IN ADMIN

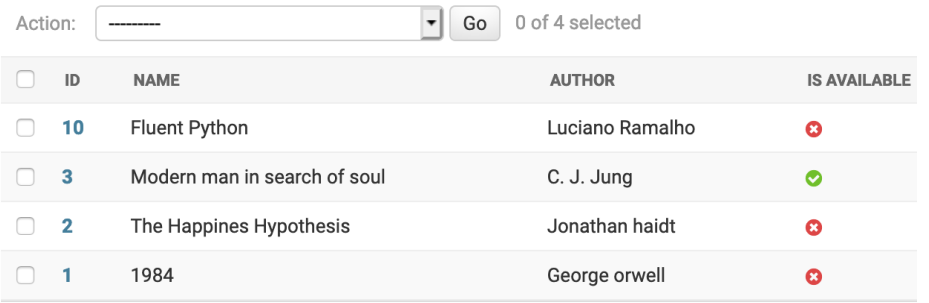
5.1 Manual Registration

Inbuilt admin interface is one the most powerful & popular feature of Django. Once we create the models, we need to register them with admin, so that it can read schema and populate interface for it.

Let us register Book model in the admin interface.

```
# file: library/book/admin.py

from django.apps import apps

from book.models import Book


class BookAdmin(admin.ModelAdmin):
    list_display = ('id', 'name', 'author')


admin.site.register(Book, BookAdmin)
```

Now, we can see the book model in admin.



If the django project has too many models to be registered in admin or if it has a legacy database where all tables need to be registered in admin, then adding all those models to admin becomes a tedious task.

5.2 Auto Registration

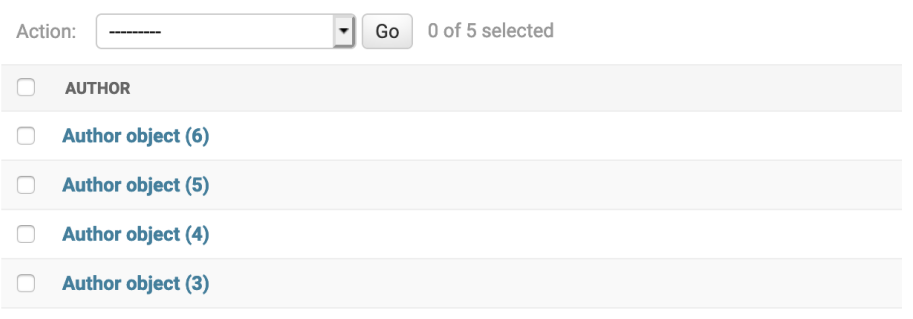
To automate this process, we can programatically fetch all the models in the project and register them with admin. Also, we need to ignore models which are already registered with admin as django doesn't allow regsitering same model twice.

```
from django.apps import apps


models = apps.get_models()

for model in models:
    try:
        admin.site.register(model)
    except admin.sites.AlreadyRegistered:
        pass
```

This code snippet should run after all *admin.py* files are loaded so that auto registration happends after all manually added models are registered. Django provides AppConfig.ready() to perform any initialization tasks which can be used to hook this code.

```
# file: library/book/apps.py

from django.apps import apps, AppConfig
from django.contrib import admin


class BookAppConfig(AppConfig):

    def ready(self):
        models = apps.get_models()
        for model in models:
            try:
                admin.site.register(model)
            except admin.sites.AlreadyRegistered:
                pass
```

In the admin, we can see manually registered models and automatically registered models. If we open admin page for any auto registered model, it will show something like this.



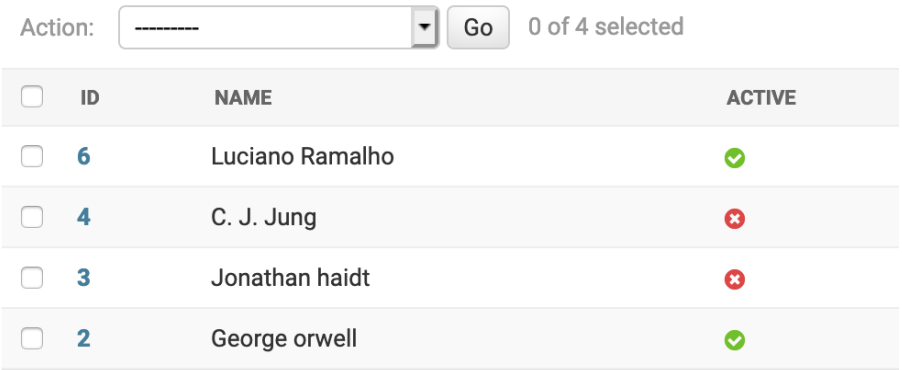
This view is not at all useful for the users who want to see the data. It will be more informative if we can show all the fields of the model in admin.

5.3 Auto Registration With Fields

To achieve that, we can create an admin class to populate model fields in *list_display*. While registering, we can use this admin class to register the model.

```
from django.apps import apps, AppConfig
from django.contrib import admin


class ListModelAdmin(admin.ModelAdmin):
    def __init__(self, model, admin_site):
        self.list_display = [field.name for field␣
↪in model._meta.fields]
        super().__init__(model, admin_site)


class BookAppConfig(AppConfig):

    def ready(self):
        models = apps.get_models()
        for model in models:
            try:
                admin.site.register(model,␣
↪ListModelAdmin)
            except admin.sites.AlreadyRegistered:
                pass
```

Now, if we look at Author admin page, it will be shown with all relevant fields.



Since we have auto registration in place, when a new model is added or columns are altered for existing models, admin interface will update accordingly without any code changes.

CHAPTER

SIX

FILTERING IN ADMIN

CHAPTER

SEVEN

CUSTOM ADMIN ACTIONS FOR QUERYSETS & INDIVIDUAL OBJECTS

7.1 Allow editing in list view

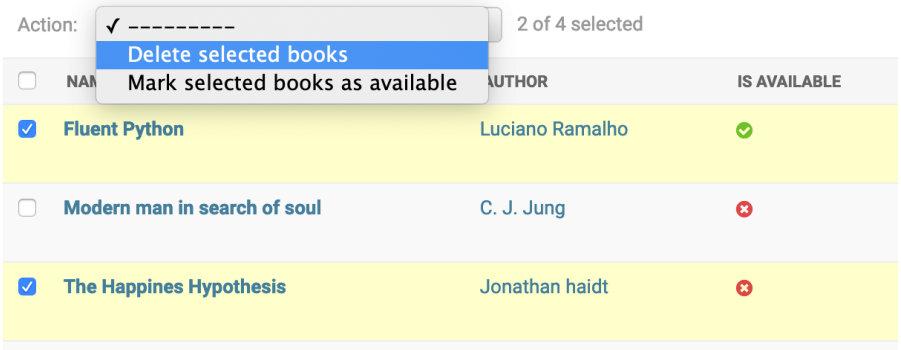
When a model is heavily used to update the content, it makes to sense to allow bulk edits on the models.

```
class BookAdmin(admin.ModelAdmin):
    list_editable = ('author',)
```

7.2 Custom Actions On Querysets

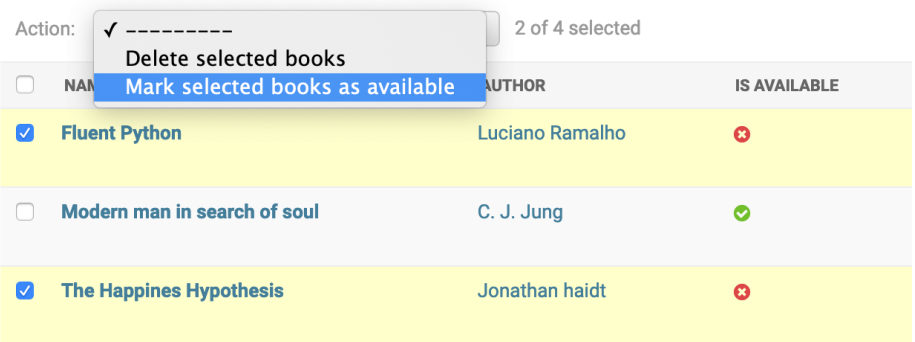
Django provides admin actions which work on a queryset level. By default, django provides delete action in the admin.

In our books admin, we can select a bunch of books and delete them.



Django provides an option to hook user defined actions to run additional actions on selected items. Let us write write a custom admin action to mark selected books as available.

```
class BookAdmin(admin.ModelAdmin):
    actions = ('make_books_available',)
    list_display = ('id', 'name', 'author')

    def make_books_available(self, modeladmin,␣
↪request, queryset):
        queryset.update(is_available=True)
    make_books_available.short_description = "Mark␣
↪selected books as available"
```



7.3 Custom Actions On Individual Objects

Custom admin actions are inefficient when taking action on an individual object. For example, to delete a single user, we need to follow these steps.

1. Select the checkbox of the object.
2. Click on the action dropdown.
3. Select "Delete selected" action.
4. Click on Go button.
5. Confirm that the objects needs to be deleted.

Just to delete a single record, we have to perform 5 clicks. That's too many clicks for a single action.

To simplify the process, we can have delete button at row level. This can be achieved by writing a function which will insert delete button for every record.

ModelAdmin instance provides a set of named URLs for CRUD operations. To get object url for a page, URL name will be *{{ app_label }}_{{ model_name }}_{{ page }}*.

For example, to get delete URL of a book object, we can call *reverse("admin:book_book_delete", args=[book_id])*. We can add a delete button with this link and add it to list_display so that delete button is available for individual objects.

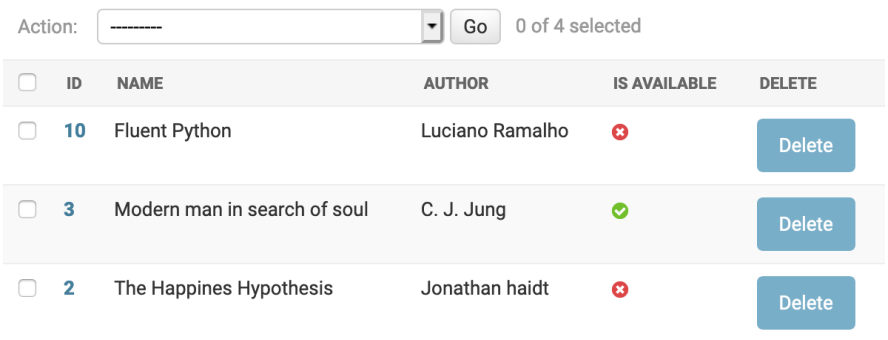
```
from django.contrib import admin
from django.utils.html import format_html

from book.models import Book


class BookAdmin(admin.ModelAdmin):
    list_display = ('id', 'name', 'author', 'is_
↪available', 'delete')
```

(continues on next page)

(continued from previous page)

```
    def delete(self, obj):
        view_name = "admin:{}_{}_delete".format(obj.
↪_meta.app_label, obj._meta.model_name)
        link = reverse(view_name, args=[book.pk])
        html = '<input type="button" onclick=
↪"location.href=\'{}\'" value="Delete" />'.
↪format(link)
        return format_html(html)
```

Now in the admin interface, we have delete button for individual objects.



To delete an object, just click on delete button and then confirm to delete it. Now, we are deleting objects with just 2 clicks.

In the above example, we have used an inbuilt model admin delete view. We can also write custom view and link those views for custom actions on individual objects. For example, we can add a button which will mark the book status to available.

In this chapter, we have seen how to write custom admin actions which work on single item as well as bulk items.

CHAPTER

EIGHT

ORM GOTCHAS

N+1 Queries

8.1 Caching

8.2 Eager evaluation

8.3 Lazy evaluation

book.author.id book.author_id

qs.exist()

.iterator()

bulk operations

bulk update wont call save or signals Runpython wont call these

get only what you need

values_list()

enable query logging

8.4 Disable full count

```
show_full_result_count = False
```

8.5 Fetch only required fields

When a model is registered in admin, django tries to fetch all the fields of the table in the query. If there are any joins involved, it fetch fields of the joined tables also. This will slow down the query when the table size is big or number of results per page is more.

To make queries faster, we can limit the queryset to fetch only required fields.

```
class BookAdmin(admin.ModelAdmin):
    def get_queryset(self, request):
        qs = super().get_queryset(request)
        qs = qs.only('id', 'name')
        return qs


admin.site.register(Book, BookAdmin)
```

CHAPTER

NINE

SECURING DJANGO ADMIN

9.1 Securing Server

9.1.1 VPN

9.1.2 FIREWALL

80/443

9.1.3 https

9.2 Securing Django

python manage.py check –deploy

9.2.1 allowed hosts

9.2.2 Disable DEBUG

9.2.3 Change default url

9.2.4 Ensuring proper ACL

9.2.5 Honeypot

https://github.com/dmpayton/django-admin-honeypot

9.2.6 2FA

https://github.com/Bouke/django-two-factor-auth

9.2.7 ENVironment

https://github.com/dizballanze/django-admin-env-notice

https://github.com/treyhunner/django-simple-history

CHAPTER

TEN

FINAL WORDS

Think about workflows.

Don’t waster too much time.