
Mastering Django Admin

ChillarAnand

Aug 29, 2021

CONTENTS

1	Preface	1
1.1	Why this book?	1
1.2	Pre requisites	1
1.3	Who should read this book?	2
1.4	Acknowledgements	2
2	The Million Dollar Admin	3
3	Better Defaults	5
3.1	Use ModelAdmin	5
3.2	Use Better Widgets	7
3.3	Better Defaults For Models	9
3.4	Navigation Menu Bar	10
4	Managing Model Relationships	13
4.1	Autocompletion For Related Fields	13
4.2	Hyperlink Related Fields	14
4.3	Related Fields In Admin List	16
5	Auto Generate Admin Interface	19
5.1	Manual Registration	19
5.2	Auto Registration	20
5.3	Auto Registration With Fields	22
5.4	Admin Generator	23

6	Filtering In Admin	25
6.1	Search Fields	25
6.2	List Filters	26
6.3	Custom List Filters	27
6.4	Custom Text Filter	28
6.5	Advanced Filters	30
7	Custom Admin Actions	33
7.1	Bulk Editing In List View	33
7.2	Custom Actions On Querysets	34
7.3	Custom Actions On Individual Objects	35
7.4	Custom Actions On Change View	37
8	Securing Django Admin	39
8.1	Admin Path	40
8.2	2 Factor Authentication	41
8.3	Environments	42
8.4	Miscellaneous	43
9	Final Words	45

PREFACE

1.1 Why this book?

In this data driven world, internal tools are often overlooked parts in companies. Without efficient tools for analytics and dashboards, cross department communications & customer communications become a bottleneck as people spend more time everyday to get some data.

There are several tools built specifically for analytics and dashboards. But if we are already using Django framework, we can just use Django Admin for most of these tasks.

In this book, we will learn how to customize Django admin for these tasks.

1.2 Pre requisites

Readers should be familiar with creating a model/view using Django. If you are new to django, complete the polls tutorial¹ provided in the official Django documentation to get familiar about Django framework.

¹ <https://docs.djangoproject.com/en/3.0/intro/tutorial01/>

1.3 Who should read this book?

Anyone who wants to learn how to customize and improve the performance of django admin.

1.4 Acknowledgements

THE MILLION DOLLAR ADMIN

Django admin was first released in 2005 and it has gone through a lot of changes since then. Still the admin interface looks clunky compared to most modern web interfaces.

Jacob Kaplan-Moss, one of the core-developers of Django estimated that it will cost 1 million dollars¹ to hire a team to rebuild admin interface from scratch. Until we get 1 million dollars to revamp the admin interface, let's look into alternate solutions.

1. Use django admin with modern themes/skins. Packages like django-grappelli², django-suit³ extend the existing admin interface and provide new skin, options to customize the UI etc.
2. Use drop-in replacements for django admin. Packages like xadmin⁴, django-admin2⁵ are a complete rewrite of django admin. Even though these packages come with lot of goodies and better defaults, they are no longer actively maintained.
3. Use separate django packages per task.
4. Write our own custom admin interface.

¹ <https://jacobian.org/2016/may/26/so-you-want-a-new-admin/>

² <https://pypi.org/project/django-grappelli/>

³ <https://pypi.org/project/django-suit/>

⁴ <https://pypi.org/project/xadmin/>

⁵ <https://pypi.org/project/django-admin2/>

We can start default admin interface or use any drop-in replacements for the admin. Even with this admin interface, we need to write custom views/reports based on business requirements.

In the next chapter, let's start with customizing admin interface.

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)
```

- ☐ BOOK
- ☐ Book object (15)
- ☐ Book object (14)
- ☐ Book object (13)
- ☐ Book object (12)

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 selected 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', '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')
```

<input type="checkbox"/>	ID	NAME	AUTHOR
<input type="checkbox"/>	1	1984	George orwell
<input type="checkbox"/>	2	The Happines Hypothesis	Jonathan haidt
<input type="checkbox"/>	3	Modern man in search of soul	C. J. Jung
<input type="checkbox"/>	10	Fluent Python	Luciano Ramalho

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.

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

    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'
```

Home » Book » Books

Select book to change ADD BOOK +

q Search

1919 1984 2009 2012 2019

Action: Go 0 of 10 selected

<input type="checkbox"/>	ID	NAME	AUTHOR	PUBLISHED DATE	IS AVAILABLE
<input type="checkbox"/>	1	1984	George orwell	Sept. 13, 1984	⊘
<input type="checkbox"/>	10	Fluent Python	Luciano Ramalho	Sept. 13, 2012	⊙
<input type="checkbox"/>	3	Modern man in search of soul	C. J. Jung	Sept. 13, 1919	⊙
<input type="checkbox"/>	2	The Happiness Hypothesis	Jonathan haidt	Sept. 13, 2009	⊘

FILTER

By is available

All

Yes

No

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 data.

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.



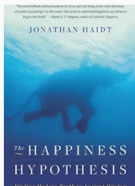

```
@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
```

(continues on next page)

(continued from previous page)

```
list_display = ('id', 'name_colored', 'thumbnail',  
→ 'author', 'published_date', 'is_available')  
  
def thumbnail(self, obj):  
    width, height = 100, 200  
    html = ''  
    return format_html(  
        html.format(url=obj.cover.url, width=width,  
→ height=height)  
    )
```

This will show thumbnail for book cover images.

<input type="checkbox"/>	ID	NAME	THUMBNAIL	AUTHOR	PUBLISHED DATE	IS AVAILABLE
<input type="checkbox"/>	1	1984		George orwell	Sept. 13, 1984	
<input type="checkbox"/>	2	The Happiness Hypothesis		Jonathan haidt	Sept. 13, 2009	

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.

CSV and Excel imports and exports

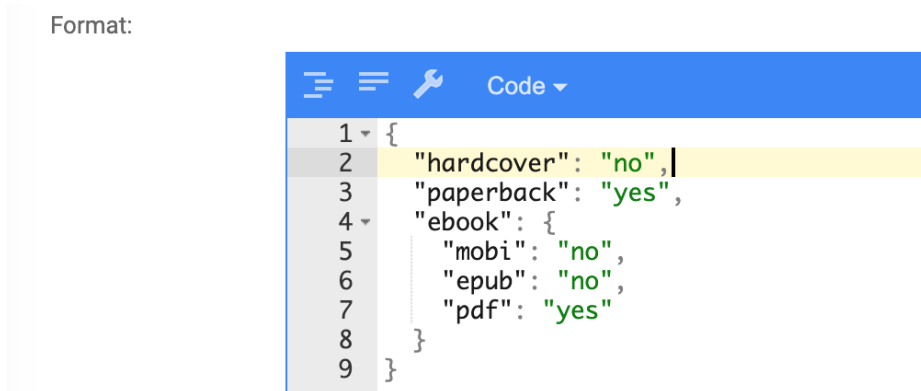
```
from django.contrib.postgres import fields  
from django_json_widget.widgets import_  
→ JSONEditorWidget
```

(continues on next page)

(continued from previous page)

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

With this, all JSONFields will use JSONEditorWidget, which makes it easy to view and edit json content.



There are a wide variety of third-party packages like django-map-widgets, django-ckeditor, django-widget-tweaks etc which provide additional widgets as well as tweaks to existing widgets.

3.3 Better Defaults For Models

We can set user friendly names instead of default names for django models in admin. We can override this in model meta options.

```
class Category(models.Model):
    class Meta:
```

(continues on next page)

(continued from previous page)

```
verbose_name = "Book Category"
verbose_name_plural = "Book Categories"
```

Model fields has an option to enter *help_text* which is useful documentation as well as help text for forms.

```
class Book(TimeAuditModel):
    is_available = models.BooleanField(
        help_text='Is the book available to buy?'
    )
    published_date = models.DateField(
        help_text='help_text="Please enter the date,
→in <em>YYYY-MM-DD</em> format.'
```

This will be shown in admin as shown below.

☒ Is available
Is the book available to buy?

Published date:

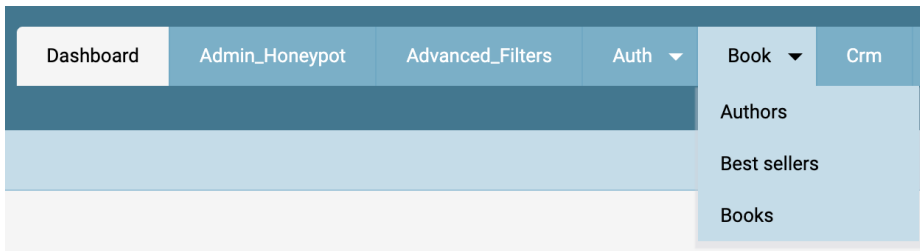
Today | 

Please enter the date in **YYYY-MM-DD** format.

3.4 Navigation Menu Bar

When user visits a specific model from the admin page, to switch to a different model user has to go back to home page and then move to the required model. This is inconvenient if user has to switch between models frequently.

To avoid this, a navigation menu bar can be added at the top as shown below, so that users can switch between models with just 1 click.



For this, we need to override *base_site.html* template with the navigation menu bar. Django provides *app_list* in the template context which has information about all apps and their models which can be used to render menu bar.

```
<ul>
    {% for app in app_list %}
        <li><a href="{ { app.app_url } }">{{ app.name }}</
    <a>
        <ul>
            {% for model in app.models %}
                <li><a href="{ { model.admin_url } }">
    <{{ model.name }}</a></li>
                {% endfor %}
            </ul>
        </li>
        {% endfor %}
    </ul>
```


MANAGING MODEL RELATIONSHIPS

4.1 Autocompletion For Related Fields

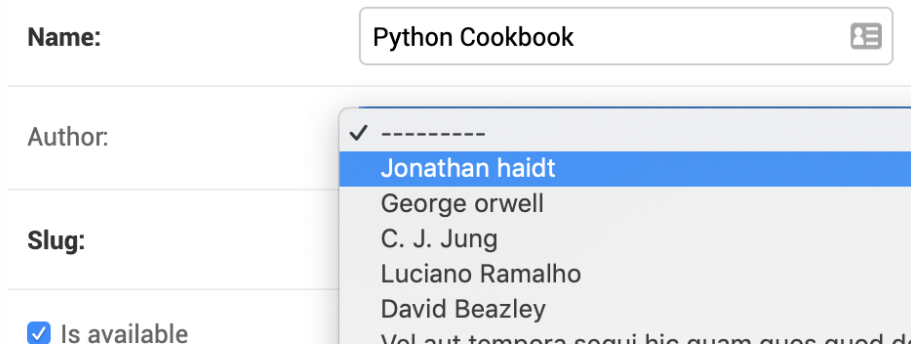
Lets us go to BookAdmin and try to add a new book.

```
from book.models import Book

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

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

By default, this will show a select box with entire authors list. Navigating this select list and finding the required author is difficult.



The screenshot shows a Django admin interface for adding a new book. The 'Name' field is filled with 'Python Cookbook'. The 'Author' field has a dropdown menu open, showing a list of authors: 'Jonathan haidt' (selected), 'George orwell', 'C. J. Jung', 'Luciano Ramalho', and 'David Beazley'. There is also a checkbox labeled 'Is available' which is checked.

To make this easier, we can provide autocomplete option for author field

so that users can search and select the required author.

```
from book.models import Book

class AuthorAdmin(admin.ModelAdmin):
    search_fields = ('name',)

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

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

For this, `ModelAdmin` provides *autocomplete_fields* option to change to `select2` autocomplete input. We should also define *search_fields* on the related admin so that search is performed on these fields.

The screenshot shows a Django Admin form for adding a new book. The 'Name' field is labeled 'Name:' and contains the text 'Python Cookbook'. Below it, the 'Author' field is labeled 'Author:' and has a dropdown menu open. The dropdown menu shows a search input with 'david' entered, and a list of suggestions: 'David Beazley' (highlighted) and 'David Jones'. To the right of the dropdown is a green plus sign. Below the 'Author' field is the 'Slug' field, which is empty. At the bottom, there is a checkbox labeled 'Is available' which is checked, and a question 'Is the book available to buy?'.

4.2 Hyperlink Related Fields

Lets browse through, BookAdmin and look at some of the books.

```
from django.contrib import admin

from .models import Book
```

(continues on next page)

(continued from previous page)

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

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

Here, book name field is linked to book change view. But author field is shown as plain text. If we notice some typo or if we have to modify author details, 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 Related Fields In Admin List

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
```

(continues on next page)

(continued from previous page)

```
from book.models import BestSeller

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

(continues on next page)

(continued from previous page)

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

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

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

By subclassing `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.

AUTO GENERATE ADMIN INTERFACE

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.

Action:

▼

Go

0 of 4 selected

<input type="checkbox"/>	ID	NAME	AUTHOR	IS AVAILABLE
<input type="checkbox"/>	10	Fluent Python	Luciano Ramalho	✖
<input type="checkbox"/>	3	Modern man in search of soul	C. J. Jung	✓
<input type="checkbox"/>	2	The Happines Hypothesis	Jonathan haidt	✖
<input type="checkbox"/>	1	1984	George orwell	✖

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.

Action: 0 of 5 selected

<input type="checkbox"/>	AUTHOR
<input type="checkbox"/>	Author object (6)
<input type="checkbox"/>	Author object (5)
<input type="checkbox"/>	Author object (4)
<input type="checkbox"/>	Author object (3)

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.

Action: Go 0 of 4 selected

<input type="checkbox"/>	ID	NAME	ACTIVE
<input type="checkbox"/>	6	Luciano Ramalho	✓
<input type="checkbox"/>	4	C. J. Jung	✗
<input type="checkbox"/>	3	Jonathan haidt	✗
<input type="checkbox"/>	2	George orwell	✓

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.

5.4 Admin Generator

The above methods will be useful to generate a pre-defined admin interface for all the models. If independent customizations are needed for the models, then we use 3rd party packages like `django-admin-generator` or `django-extensions` which can generate a fully functional admin interface by introspecting the models. Once the base admin code is ready, we can use the same for further customizations.

```
$ ./manage.py admin_generator books >> books/admin.  
↪py
```

This will generate admin interface for *books* app.

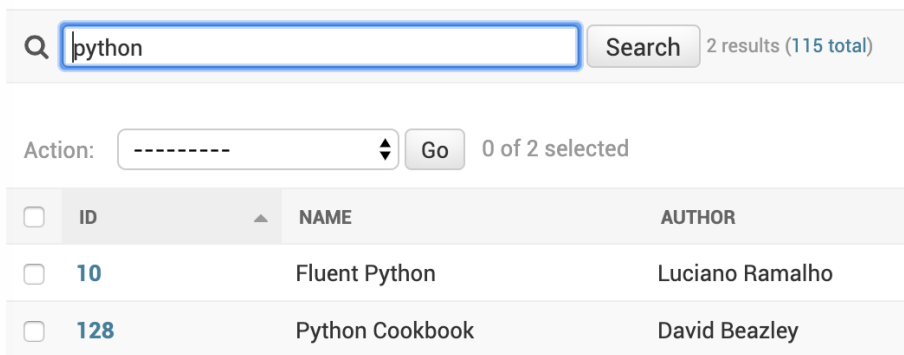
FILTERING IN ADMIN

6.1 Search Fields

Django Admin provides *search_fields* option on *ModelAdmin*. Setting this will enable a search box in list page to filter items on the model. This can perform lookup on all the fields on the model as well as related model fields.

```
class BookAdmin(admin.ModelAdmin):  
    search_fields = ('name', 'author__name')
```

Select book to change



The screenshot shows the Django Admin interface for a 'Book' model. At the top, there is a search bar with a magnifying glass icon, containing the text 'python'. To the right of the search bar is a 'Search' button and a text indicating '2 results (115 total)'. Below the search bar, there is an 'Action:' dropdown menu with a downward arrow, a 'Go' button, and a text indicating '0 of 2 selected'. Below this, there is a table with three columns: 'ID', 'NAME', and 'AUTHOR'. The table contains two rows of data.

<input type="checkbox"/>	ID	NAME	AUTHOR
<input type="checkbox"/>	10	Fluent Python	Luciano Ramalho
<input type="checkbox"/>	128	Python Cookbook	David Beazley

When the number of items in *search_fields* becomes increases, query becomes quite slow as it does a case-insensitive search of all the search

terms against all the `search_fields`. For example a search for *python for data analysis* translates to this SQL clause.

```
WHERE
(name ILIKE '%python%' OR author.name ILIKE '%python
→%')
AND (name ILIKE '%for%' OR author.name ILIKE '%for%
→')
AND (name ILIKE '%data%' OR author.name ILIKE '%data
→%')
AND (name ILIKE '%analysis%' OR author.name ILIKE '
→%analysis%')
```

6.2 List Filters

Django also provides *list_filter* option on *ModelAdmin*. We can add required fields to *list_filter* which generate corresponding filters on the right panel of the admin page with all the possible values.

```
class BookAdminFilter(admin.ModelAdmin):
    list_display = ('id', 'author', 'published_date
→', 'is_available', 'cover')
    list_filter = ('is_available',)
```

ADD BOOK +

FILTER

By is available

All

Yes

No

6.3 Custom List Filters

We can also write custom filters so that we can set calculated fields and add filters on top of them.

```
class CenturyFilter(admin.SimpleListFilter):
    title = 'century'
    parameter_name = 'published_date'

    def lookups(self, request, model_admin):
        return (
            (21, '21st century'),
            (20, '20th century'),
        )

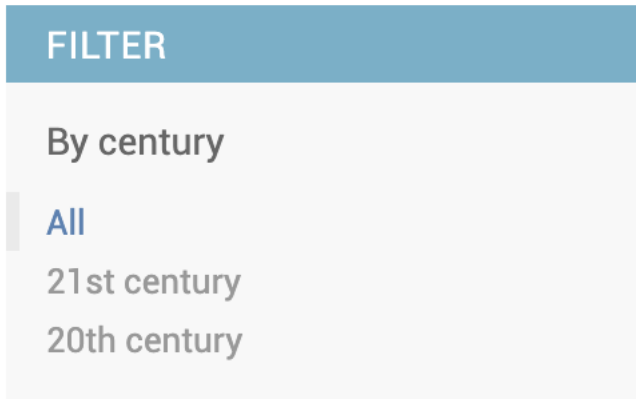
    def queryset(self, request, queryset):
        value = self.value()
        if not value:
            return queryset
        start = (int(value) - 1) * 100
        end = start + 99
```

(continues on next page)

(continued from previous page)

```
return queryset.filter(published_date__year_  
→_gte=start, published_date__year__lte=end)
```

ADD BOOK +



FILTER

By century

- All
- 21st century
- 20th century

6.4 Custom Text Filter

Here the number of choices are limited. But in some cases where the choices are hundred or more, it is better to display a text input instead of choices.

Let's write a custom filter to filter books by published year. Let's write an input filter

```
class PublishedYearFilter(admin.SimpleListFilter):  
    title = 'published year'  
    parameter_name = 'published_date'  
    template = 'admin_input_filter.html'  
  
    def lookups(self, request, model_admin):  
        return ((None, None),)
```

(continues on next page)

(continued from previous page)

```
def choices(self, changelist):
    query_params = changelist.get_filters_
    ↪params()
    query_params.pop(self.parameter_name, None)
    all_choice = next(super().
    ↪choices(changelist))
    all_choice['query_params'] = query_params
    yield all_choice

def queryset(self, request, queryset):
    value = self.value()
    if value:
        ↪return queryset.filter(published_date__
    ↪year=value)
```

This will show in admin like this.

```
{% load i18n %}

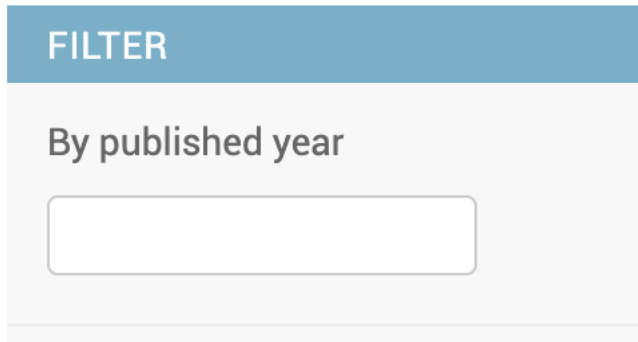
<h3>{% blocktrans with filter_title=title %} By {{
    ↪filter_title }} {% endblocktrans %}</h3>
<ul>
    <li>
        {% with choices.0 as all_choice %}
            <form method="GET">
                <input type="text" name="{{ spec.
    ↪parameter_name }}" value="{{ spec.qvalue|default_
    ↪if_none:"" }}" />
                <input class="btn btn-info" type=
    ↪"submit" value="{% trans 'Apply' %}">
                {% if not all_choice.selected %}
                    <button type="button" class=
    ↪"btn btn-info"><a href="{{ all_choice.query_
    ↪string }}">Clear</a></button>
                {% endif %}
            </form>
        {% endwith %}
```

(continues on next page)

(continued from previous page)

```
</li>  
</ul>
```

ADD BOOK +



<https://stackoverflow.com/a/20588975/2698552>

6.5 Advanced Filters

All the above methods will be useful only to a certain extent. Beyond that, there are 3rd party packages like *django-advanced-filters* which advanced filtering abilities.

To setup the package

- Install the package with *pip install django-advanced-filters*.
- Add *advanced_filters* to `INSTALLED_APPS`.
- Add `url(r'^advanced_filters/', include('advanced_filters.urls'))` to project `urlpatterns`.
- Run `python manage.py migrate`.

Once the setup is completed, we can add ``

```
from advanced_filters.admin import AdminAdvancedFiltersMixin
↳ AdminAdvancedFiltersMixin

class BookAdAdminFilter(AdminAdvancedFiltersMixin, admin.ModelAdmin):
↳ admin.ModelAdmin):
    list_display = ('id', 'name', 'author',
↳ 'published_date', 'is_available', 'name')
    advanced_filter_fields = ('name', 'published_
↳ date', 'author', 'is_available')
```

In the admin page, a popup like this will be shown to apply advanced filters.

Create advanced filter: ×

Title:

FIELD	OPERATOR	VALUE	NEGATE	DELETE
<input type="text" value="Author"/>	<input type="text" value="Equals"/>	<input type="text"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="text" value="Published date"/>	<input type="text" value="Equals"/>	<input type="text"/>	<input type="checkbox"/>	<input type="checkbox"/>

[Add another filter](#)

A simple filter can be created to filter all the books that were published between 1980 to 1990 which have a rating more than 3.75 and number of pages is not more than 100. This filter can be named and saved for later use.

CUSTOM ADMIN ACTIONS

7.1 Bulk Editing In List View

For data cleanup and heavy content updates, bulk editing on a model makes workflow easier. Django provides *list_editable* option to make selected fields editable in the list view itself.

```
class BestsellerAdmin(RelatedFieldAdmin):  
    list_display = ('id', 'book', 'year', 'rank')  
    list_editable = ('book', 'year', 'rank')
```

This allows to edit the above mentioned fields as shown.

Action: 0 of 4 selected

<input type="checkbox"/>	ID	BOOK	YEAR	RANK
<input type="checkbox"/>	6	<input type="text" value="Fluent Python"/> <input type="button" value="x"/> <input type="button" value="+"/> <input type="button" value="x"/>	<input type="text" value="2000"/>	<input type="text" value="4"/>
<input type="checkbox"/>	5	<input type="text" value="1984"/> <input type="button" value="x"/> <input type="button" value="+"/> <input type="button" value="x"/>	<input type="text" value="2018"/>	<input type="text" value="3"/>
<input type="checkbox"/>	2	<input type="text" value="The Happines Hypothesis"/> <input type="button" value="x"/> <input type="button" value="+"/> <input type="button" value="x"/>	<input type="text" value="2019"/>	<input type="text" value="2"/>
<input type="checkbox"/>	1	<input type="text" value="Modern man in search of soul"/> <input type="button" value="x"/> <input type="button" value="+"/> <input type="button" value="x"/>	<input type="text" value="2020"/>	<input type="text" value="1"/>

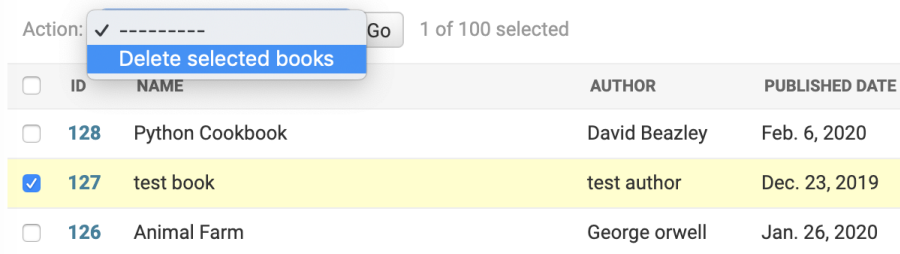
4 best sellers

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.

Select book to change



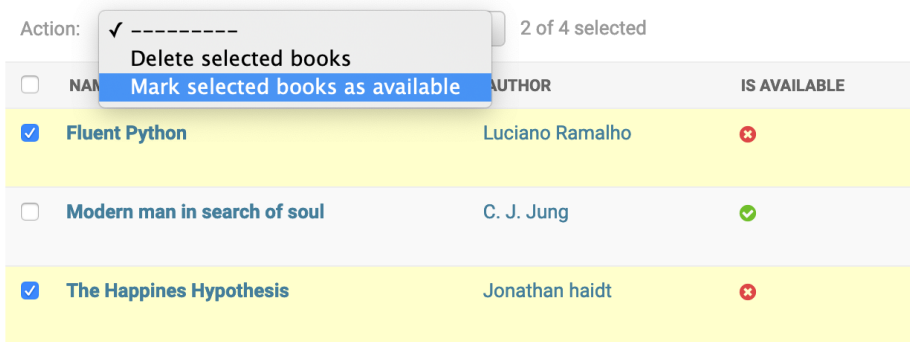
The screenshot shows the Django Admin interface for a 'books' model. At the top, there is a form with an 'Action:' dropdown menu, a 'Go' button, and a status '1 of 100 selected'. The dropdown menu is open, showing a list of actions, with 'Delete selected books' highlighted in blue. Below the form is a table with the following columns: 'ID', 'NAME', 'AUTHOR', and 'PUBLISHED DATE'. The table contains three rows of data. The first row has ID 128, name 'Python Cookbook', author 'David Beazley', and published date 'Feb. 6, 2020'. The second row has ID 127, name 'test book', author 'test author', and published date 'Dec. 23, 2019'. The third row has ID 126, name 'Animal Farm', author 'George orwell', and published date 'Jan. 26, 2020'. The second row is highlighted in yellow, and its checkbox is checked.

	ID	NAME	AUTHOR	PUBLISHED DATE
<input type="checkbox"/>	128	Python Cookbook	David Beazley	Feb. 6, 2020
<input checked="" type="checkbox"/>	127	test book	test author	Dec. 23, 2019
<input type="checkbox"/>	126	Animal Farm	George orwell	Jan. 26, 2020

Django provides an option to hook user defined actions to run additional actions on selected items. Let us 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"
```



Instead of having custom actions in the drop down, we can put dedicated buttons for most frequently used actions to reduce number of clicks needed to perform an action.

<https://github.com/crccheck/django-object-actions>

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')

    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.

Action:

▼

Go

0 of 4 selected

<input type="checkbox"/>	ID	NAME	AUTHOR	IS AVAILABLE	DELETE
<input type="checkbox"/>	10	Fluent Python	Luciano Ramalho	✖	<div>Delete</div>
<input type="checkbox"/>	3	Modern man in search of soul	C. J. Jung	✔	<div>Delete</div>
<input type="checkbox"/>	2	The Happines Hypothesis	Jonathan haidt	✖	<div>Delete</div>

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.

7.4 Custom Actions On Change View

When users want to conditionally perform a custom action when an object gets modified, custom action buttons can be provided on the change view. For example, when a best seller is updated, notify the author of the best seller via an email.

We can override *change_form.html* to include a button for custom action.

```
{% extends 'admin/change_form.html' %}

{% block submit_buttons_bottom %}
    {{ block.super }}
    <div class="submit-row">
        <input type="submit" value="Notify_
↪Author" name="notify-author">
    </div>
{% endblock %}
```

In the admin view, we have to override *response_change* to handle the submit button press.

```
class BestsellerAdmin(admin.ModelAdmin):
    change_form_template = "bestseller_changeform.
↪html"

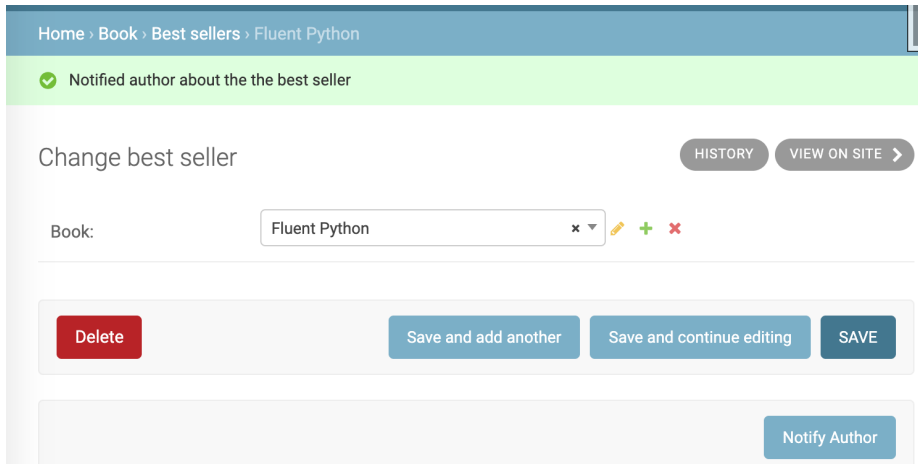
    def response_change(self, request, obj):
        if "notify-author" in request.POST:
            send_best_seller_email(obj)
            self.message_user(request, "Notified_
↪author about the best seller")
```

(continues on next page)

(continued from previous page)

```
        return HttpResponseRedirect(request.  
→path_info)  
        return super().response_change(request, obj)
```

This will show a button on the change form as shown below.



Add confirmation page for potentially dangerous actions.

There is a 3rd party package `django-admin-row-actions`, which provides a mixin to create custom admin actions.

<https://github.com/DjangoAdminHackers/django-admin-row-actions>

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

SECURING DJANGO ADMIN

Once the Django Admin is up and running with required functionality, it is necessary to ensure that it doesn't not have unattended access.

First, the server infrastructure needs to be secured. This is topic itself can be written as a separate book. How to secure a linux server¹ guide provides detailed instructions on how to secure a server.

Next, we need to ensure our Django application is secure. Django provides documentation² on how to secure a django powered site.

Django provides system check to inspect entire code base and report common issues. We can run this command with *--deploy* which activates additional checks for deployment.

```
$ python manage.py check --deploy
```

In this chapter let us focus at Admin related security measures to make it secure.

¹ <https://github.com/imthenachoman/How-To-Secure-A-Linux-Server>

² <https://docs.djangoproject.com/en/3.0/topics/security/>

8.1 Admin Path

Most of the django sites use `/admin/` as the default path for admin interface. This needs to be changed to a different path.

```
url(r'^secret-path/', admin.site.urls)
```

We can write a system check to check if `/admin/` path is used and raise an error.

```
from django.conf import settings
from django.core.checks import Error, Tags, register
from django.urls import resolve

@register(Tags.security, deploy=True)
def check_admin_path(app_configs, **kwargs):
    errors = []
    try:
        default_admin = resolve('/admin/')
    except Resolver404:
        default_admin = None
    if default_admin:
        msg = 'Found admin in default "/admin/" path'
        hint = 'Route admin via different url'
        errors.append(Error(msg, hint))

    return errors
```

Instead of removing admin, We can also setup a honeypot at the default path which will serve a fake login page. To install honeypot³, run `pip install django-admin-honeypot` with pip, add `admin_honeypot` to `INSTALLED_APPS` and set default path to honeypot path in urls.

```
url(r'^admin/', include('admin_honeypot.urls',
↳ namespace='admin_honeypot'))
```

³ <https://github.com/dmpayton/django-admin-honeypot>

Now, we can track all the login attempts on the honeypot admin from the admin page.

Home › Admin_Honeypot › Login attempts

Select login attempt to change

username	ip address	session	timestamp
admin	127.0.0.1	None	Jan. 25, 2020, 1:08 a.m.
root	127.0.0.1	None	Jan. 25, 2020, 1:17 a.m.

8.2 2 Factor Authentication

To make Admin more secure, we can enable 2 step verification where user has to provide 2 different authentication factors, one is password and the other is OTP generated from user mobile.

For this, we can use *django-otp*⁴ package and create a custom admin config to replace default admin site.

Install the package with *pip install django-otp*, add *django_otp*, *django_otp.plugins.otp_totp* to installed apps and run *./manage.py migrate*.

In the admin page, under *OTP_TOTP* section add new device so that we can generate OTP for the admin page.

Create 2 files *admin.py* & *apps.py* in the project package to create custom admin config for *OTPAdminSite* and set it as default.

```
from django_otp.admin import OTPAdminSite
```

(continues on next page)

⁴ <https://pypi.org/project/django-otp/>

(continued from previous page)

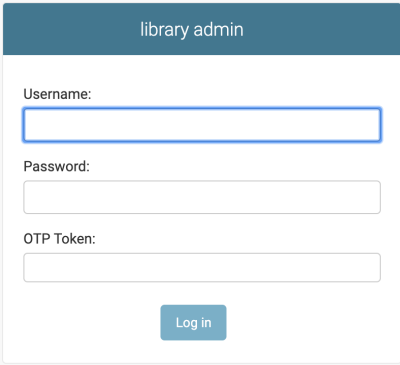
```
class LibraryOTPAdminSite (OTPAdminSite):  
    pass
```

```
from django.contrib.admin.apps import AdminConfig  
  
class LibraryAdminConfig (AdminConfig):  
    default_site = 'library.admin.  
→LibraryOTPAdminSite'
```

In the *INSTALLED_APPS*, replace admin with custom config.

```
INSTALLED_APPS = [  
    # 'django.contrib.admin',  
    'library.apps.LibraryAdminConfig',  
]
```

Now the admin login page will show OTP login form.



8.3 Environments

When the django app is deployed in multiple environments, it is important to distinguish those environments visually so that admin users accidentally don't modify data in production environments. For this we can override the base template with a custom template.

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

8.4 Miscellaneous

If you have user groups and permissions, it is important to set permissions on object level.

When using *ModelAdmin.get_urls()* to extend urls, Django by default doesn't do any permission checks and the view is accessible to public. Ensure that these views are secure by wrapping them with *admin_view*.

```
class BookAdmin(admin.ModelAdmin):
    def get_urls(self):
        urls = super().get_urls()
        book_urls = [
            path('book_char_data/', self.admin_site.
↪admin_view(self.chart_data))
        ]
        return book_urls + urls
```


FINAL WORDS

This short book is written to customize admin interface so that custom views, reports, analytics are generated in the admin itself.

To provide feedback about this book, please write to *chillar@avilpage.com*.