Mastering Django Admin

ChillarAnand

Jan 12, 2020

CONTENTS

CHAPTER

ONE

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 work between departments becomes a bottleneck as people are handling one-off requests. This becomes a bottleneck and the inter communication efficiency will decrease.

Inbuilt admin interface

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

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 aNot good ux

Jacob Kaplan-Moss, one of the core-developers of Django estimated that it will cost 1 million dollars 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.

There are few opensource 3rd party packages like xadmin, django-admin2 which aimed to provide drop-in replacement for django. Even though these packages provide additional features to extend admin interface and provide better UI, they are not well maintained as django itself.

mental models

Revamp

reduce cost of maintainance and development

people come to django becase of admin

more visual and responsive

CHAPTER

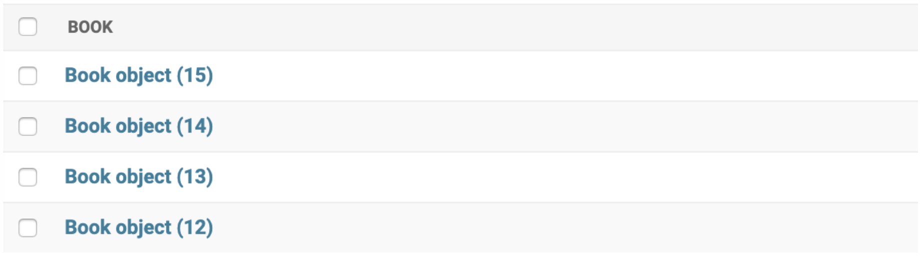
THREE

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)
```



BOOK

Book object (15)

Book object (14)

Book object (13)

Book object (12)

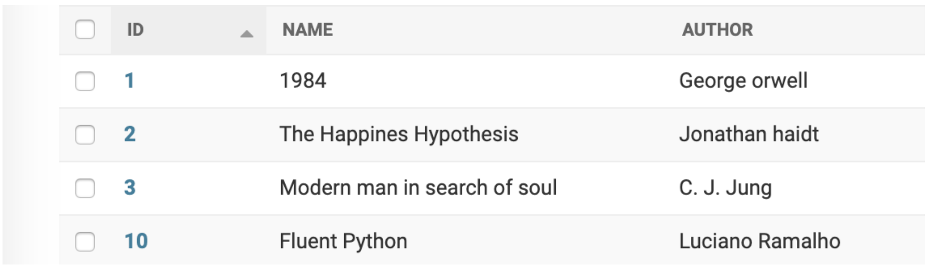
Django provides ModelAdmin[1] class which represents a model in admin. We can use the following options to make the admin interface informative and easy to use.

- *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 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.

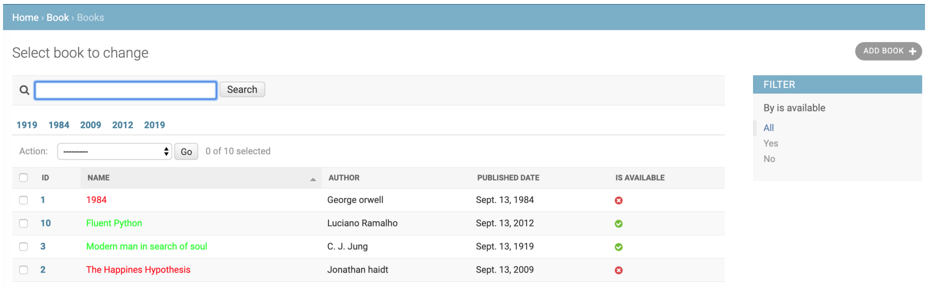
```
from book.models import Book
from django.contrib import admin


@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
    list_display = ('id', 'name', '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')
```



	ID	NAME	AUTHOR
☐	1	1984	George orwell
☐	2	The Happines Hypothesis	Jonathan haidt
☐	3	Modern man in search of soul	C. J. Jung
☐	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'
```



ID	NAME	AUTHOR	PUBLISHED DATE	IS AVAILABLE
1	1984	George orwell	Sept. 13, 1984	✗
10	Fluent Python	Luciano Ramalho	Sept. 13, 2012	✓
3	Modern man in search of soul	C. J. Jung	Sept. 13, 1919	✓
2	The Happines Hypothesis	Jonathan haidt	Sept. 13, 2009	✗

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.

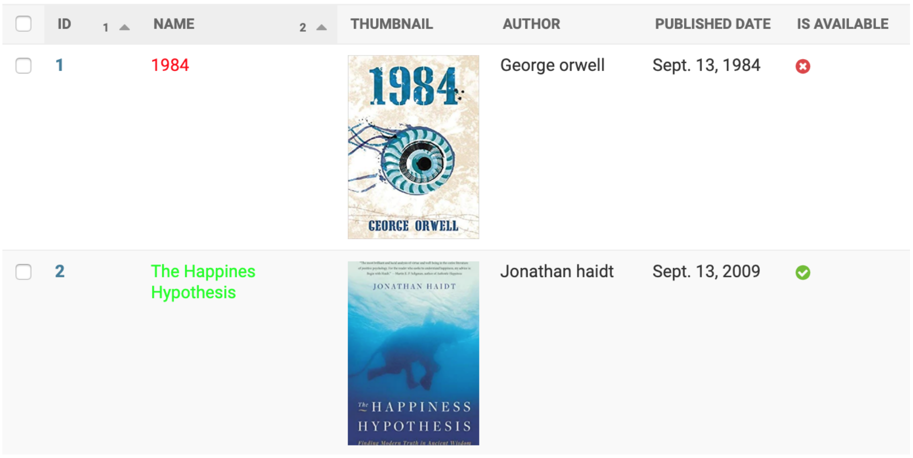
```
@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
```

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

This will show thumbnail for book cover images.



	ID 1	NAME 2	THUMBNAIL	AUTHOR	PUBLISHED DATE	IS AVAILABLE
☐	1	1984		George orwell	Sept. 13, 1984	✗
☐	2	The Happines 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.

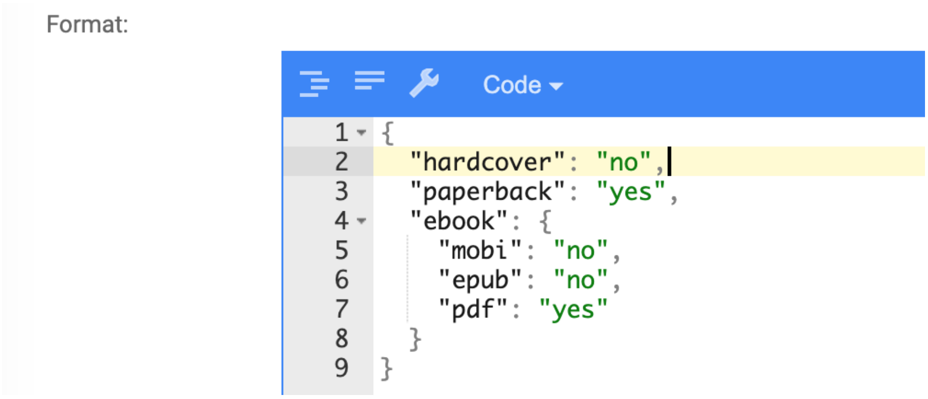
```
from django.contrib.postgres import fields
from django_json_widget.widgets import␣
↪JSONEditorWidget

@admin.register(Book)
```

(continues on next page)

(continued from previous page)

```python
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.



```
{
  "hardcover": "no",
  "paperback": "yes",
  "ebook": {
    "mobi": "no",
    "epub": "no",
    "pdf": "yes"
  }
}
```

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

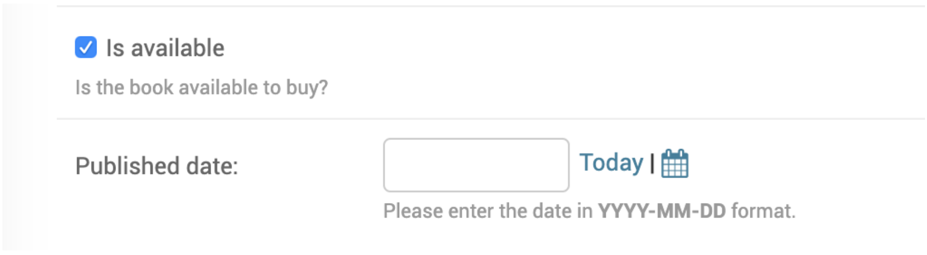
We can set user friendly names instead of default names for django models in admin. We can override this in model meta options.

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

CHAPTER

FOUR

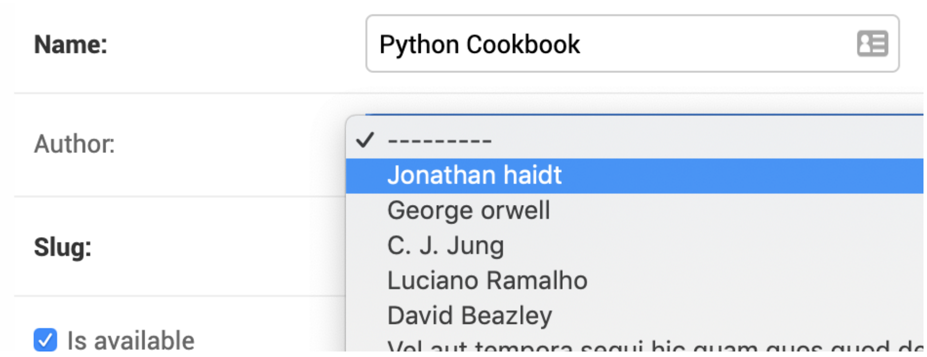
MANAGING MODEL RELATIONSHIPS

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.



Name: Python Cookbook

Author:
✓ ---------
Jonathan haidt
George orwell
C. J. Jung
Luciano Ramalho
David Beazley

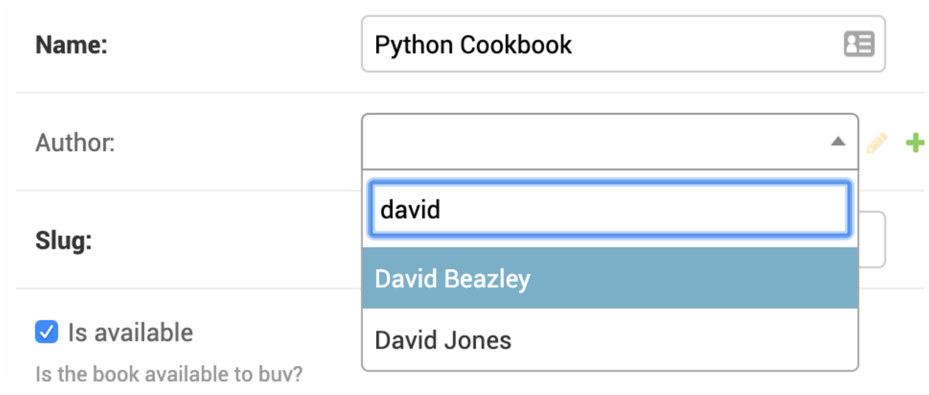
Slug:

✓ Is available

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.



Name: Python Cookbook

Author:

david

David Beazley

David Jones

Slug:

✓ Is available

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 liked 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 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 GENERATE ADMIN INTERFACE

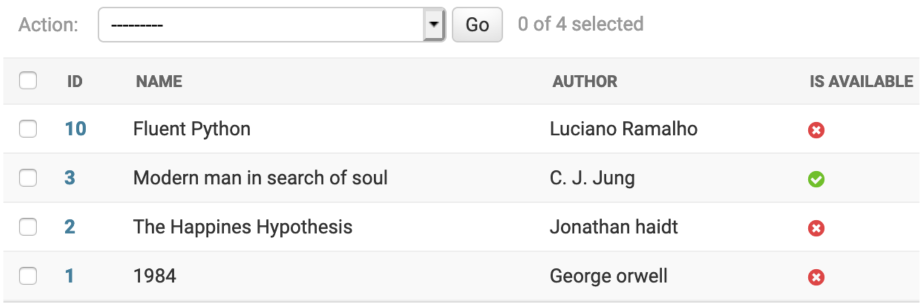
5.1 Manual Registration

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.



ID	NAME	AUTHOR	IS AVAILABLE
10	Fluent Python	Luciano Ramalho	✗
3	Modern man in search of soul	C. J. Jung	✓
2	The Happines Hypothesis	Jonathan haidt	✗
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

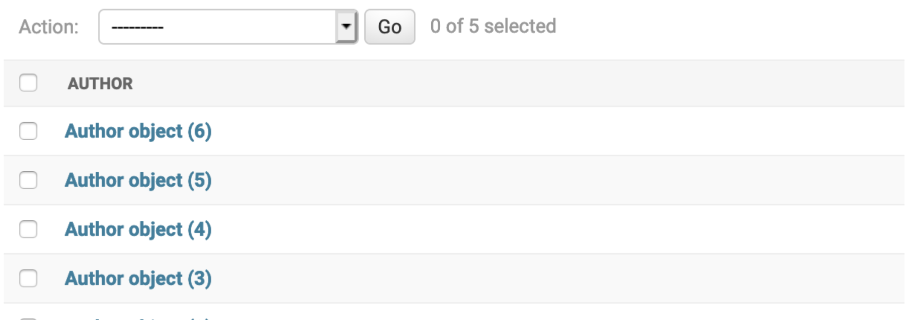
To automate this process, we can programatically fetch all the models in the project and register them with admin. Also, we need to ignore models which are already registered with admin as django doesn't allow regsitering same model twice.

```
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.

```python
# 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.



AUTHOR
Author object (6)
Author object (5)
Author object (4)
Author object (3)

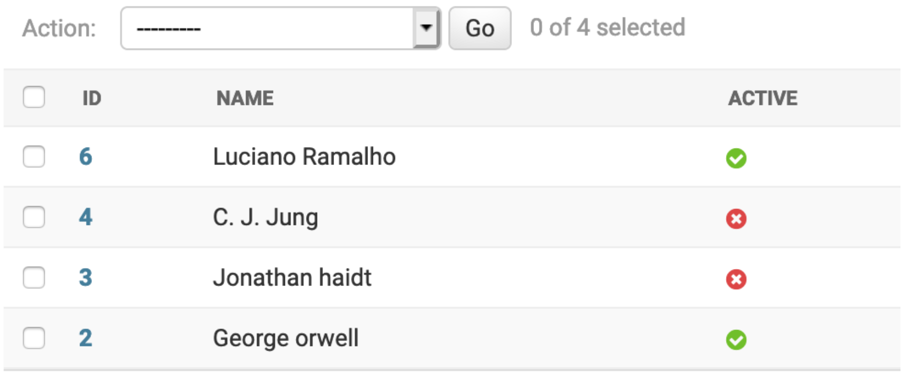
This view is not at all useful for the users who want to see the data. It will be more informative if we can show all the fields of the model in admin.

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.



ID	NAME	ACTIVE
6	Luciano Ramalho	✓
4	C. J. Jung	✗
3	Jonathan haidt	✗
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 futher customizations.

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

This will generate admin interface for *books* app.

CHAPTER

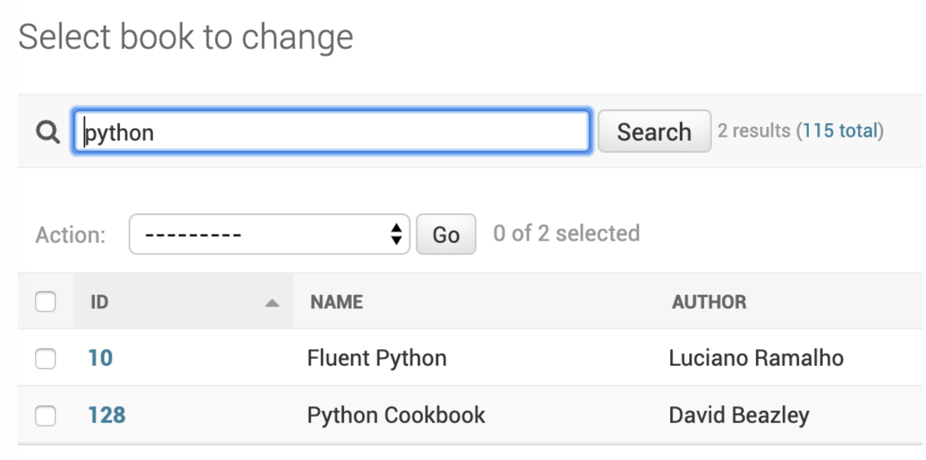
SIX

FILTERING IN ADMIN

6.1 Search Fields

Django Admin provies *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

python

2 results (115 total)

Action: --------- 0 of 2 selected

ID	NAME	AUTHOR
10	Fluent Python	Luciano Ramalho
128	Python Cookbook	David Beazley

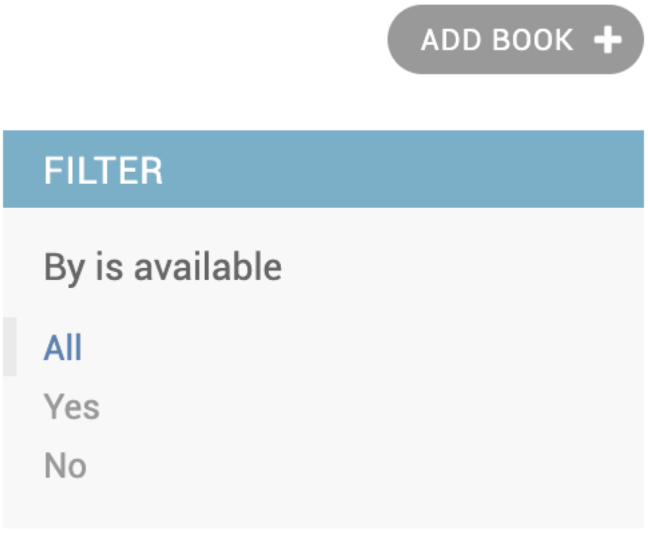
When the number of items in search_fields becomes increases, query becomes quite slow as it does a case-insensitive search of all the search terms against all the search_fields. For example a search for *python for data analysis* translates to this SQL caluse.

```
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',)
```



6.3 Custom List Filters

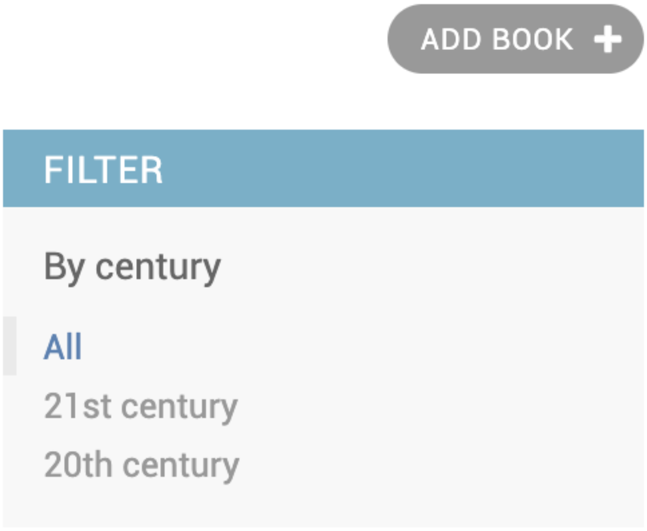
We can also write custom filters so that we can set calculated fields and add filters on top of them.

```
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)
```



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)
```

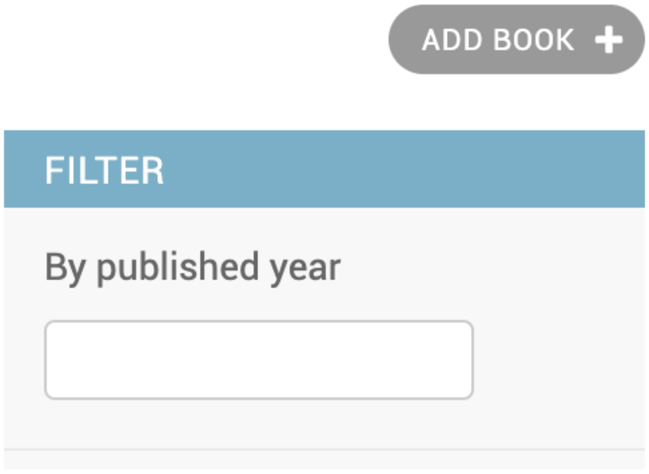
This will show in admin like this.

```
{% 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>
```



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 abilites.

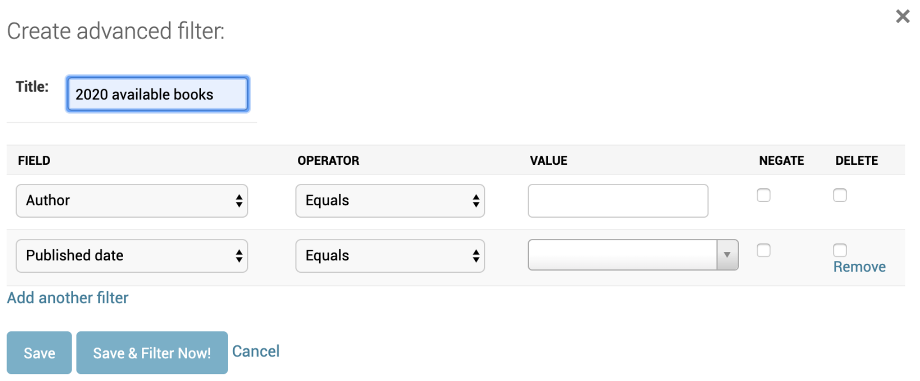
To setup the package

- 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 urlconf.
- Run *python manage.py migrate*.

Once the setup is completed, we can add ``

```
from advanced_filters.admin import␣
↪AdminAdvancedFiltersMixin

class BookAdAdminFilter(AdminAdvancedFiltersMixin,␣
↪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 filers.


Create advanced filter:

Title: 2020 available books

FIELD	OPERATOR	VALUE	NEGATE	DELETE
Author	Equals		☐	☐
Published date	Equals		☐	☐ Remove

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.

https://github.com/modlinltd/django-advanced-filters

CHAPTER

SEVEN

CUSTOM ADMIN ACTIONS

7.1 Allow editing in list view

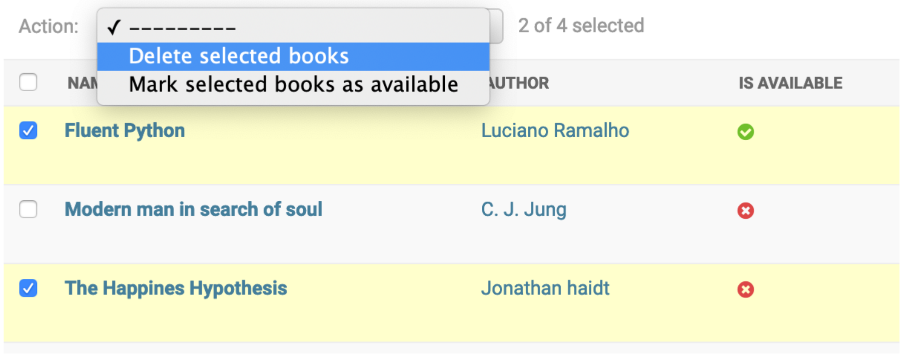
When a model is heavily used to update the content, it makes to sense to allow bulk edits on the models.

```
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.

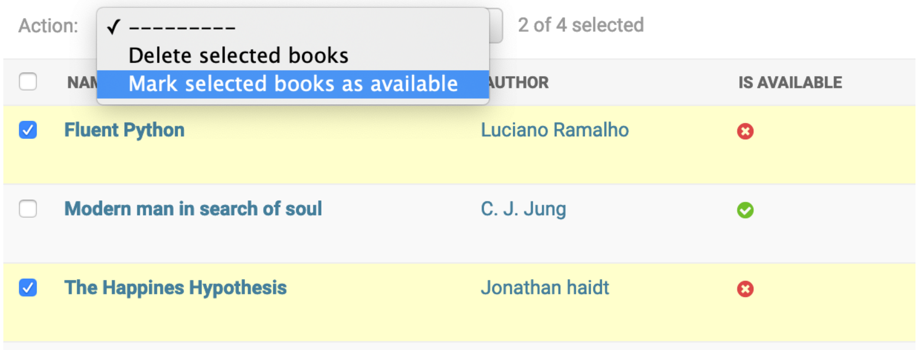
In our books admin, we can select a bunch of books and delete them.



	NAME	AUTHOR	IS AVAILABLE
☑	Fluent Python	Luciano Ramalho	✓
☐	Modern man in search of soul	C. J. Jung	✗
☑	The Happines Hypothesis	Jonathan haidt	✗

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"
```



	NAME	AUTHOR	IS AVAILABLE
☑	Fluent Python	Luciano Ramalho	✗
☐	Modern man in search of soul	C. J. Jung	✓
☑	The Happines Hypothesis	Jonathan haidt	✗

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.

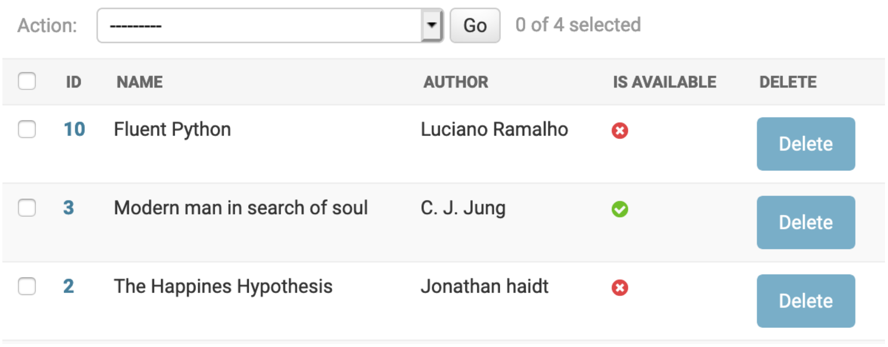
```
from django.contrib import admin
from django.utils.html import format_html

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.



ID	NAME	AUTHOR	IS AVAILABLE	DELETE
10	Fluent Python	Luciano Ramalho	✗	Delete
3	Modern man in search of soul	C. J. Jung	✓	Delete
2	The Happines Hypothesis	Jonathan haidt	✗	Delete

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

DASHBOARDS

https://github.com/byashimov/django-controlcenter

CHAPTER

NINE

SECURING DJANGO ADMIN

There are several security measures that needs to be taken care at system level as well as django level to make sure Django apps are secure. In this chapter let us specifically look at admin related things which needs to be taken care.

9.1 Admin Path

Most of the django sites use */admin/* as the path for admin interface. This needs to be changed to a different path. If you want, you can setup a honeypot server at the default path to see the attacks on your admin site.

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

9.2 ACL

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

9.3 2FA

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

9.4 ENVironment

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

CHAPTER

TEN

FINAL WORDS

Think about workflows.

Don’t waster too much time.