
Mastering Django Admin

ChillarAnand

Dec 28, 2019

CONTENTS

1	Preface	1
1.1	Why this book?	1
1.2	Who should read this book?	1
1.3	Acknowledgements	1
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
4	Managing Model Relationships	11
4.1	Autocompletion For Related Fields	11
4.2	Hyperlink Related Fields	12
4.3	Related Fields In Admin List	14
5	Auto Generate Admin Interface	17
5.1	Manual Registration	17
5.2	Auto Registration	18
5.3	Auto Registration With Fields	20
5.4	Admin Generator	21
6	Filtering In Admin	23
7	Custom Admin Actions	25

7.1	Allow editing in list view	25
7.2	Custom Actions On Querysets	25
7.3	Custom Actions On Individual Objects	27
8	ORM Gotchas	29
8.1	Caching	29
8.2	Eager evaluation	29
8.3	Lazy evaluation	29
8.4	Disable full count	30
8.5	Fetch only required fields	30
9	Dashboards	31
10	Securing Django Admin	33
10.1	Securing Server	33
10.2	Securing Django	33
11	Final Words	35

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 Acknowledgements

Krace Kumar

Haris Ibrahim

Tim Graham,<https://techytim.com/>

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

Haki Banita

<https://hakibenita.com/>

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 because of admin

more visual and responsive

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 +

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="radio"/>
<input type="checkbox"/>	10	Fluent Python	Luciano Ramalho	Sept. 13, 2012	<input checked="" type="radio"/>
<input type="checkbox"/>	3	Modern man in search of soul	C. J. Jung	Sept. 13, 1919	<input checked="" type="radio"/>
<input type="checkbox"/>	2	The Happiness Hypothesis	Jonathan haidt	Sept. 13, 2009	<input type="radio"/>

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.



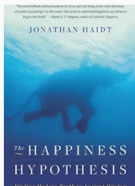

```
@admin.register(Book)
class BookAdmin(admin.ModelAdmin):
```

(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.

```
from django.contrib.postgres import fields  
from django_json_widget.widgets import  
→ JSONEditorWidget  
  
@admin.register(Book)
```

(continues on next page)

(continued from previous page)

```
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

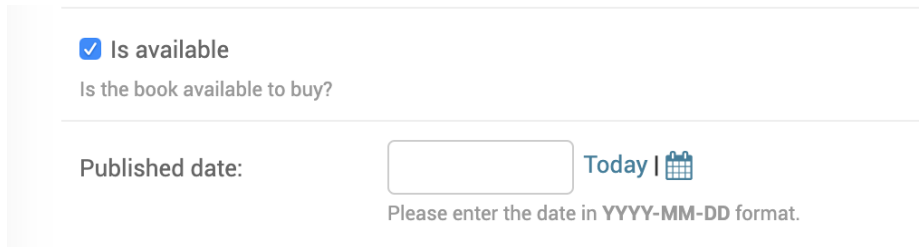
We can set user friendly names instead of default names for django models in admin. We can override this in model meta options.

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



The screenshot shows a Django Admin form for the 'Book' model. It features a checkbox labeled 'Is available' with a blue checkmark, followed by the help text 'Is the book available to buy?'. Below this is a 'Published date:' label next to a date input field. To the right of the input field is a 'Today' button with a calendar icon. Below the input field, a message states 'Please enter the date in YYYY-MM-DD format.'

MANAGING MODEL RELATIONSHIPS

4.1 Autocompletion For Related Fields

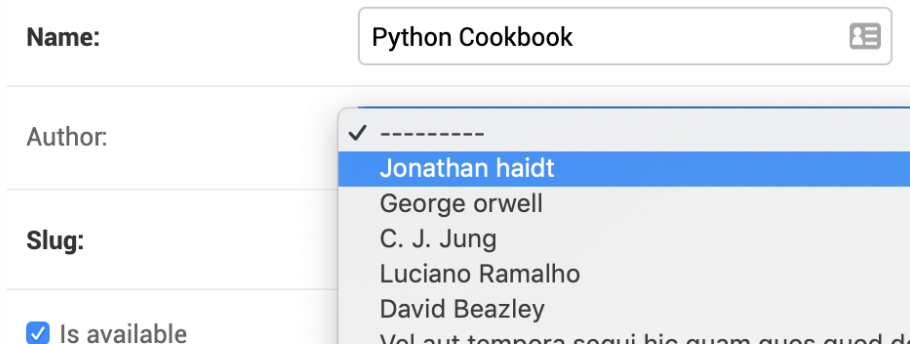
Lets us go to BookAdmin and try to add a new book.

```
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 form 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'. The 'Slug' field is empty. There is 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.

Name: Python Cookbook

Author: david

Slug:

☒ Is available

Is the book available to buy?

David Beazley

David Jones

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.__getattr__(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.

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

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` 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

Django Admin provides *search_fields* options 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')
```

When the number of items in *search_fields* becomes more, query becomes

CUSTOM ADMIN ACTIONS

7.1 Allow editing in list view

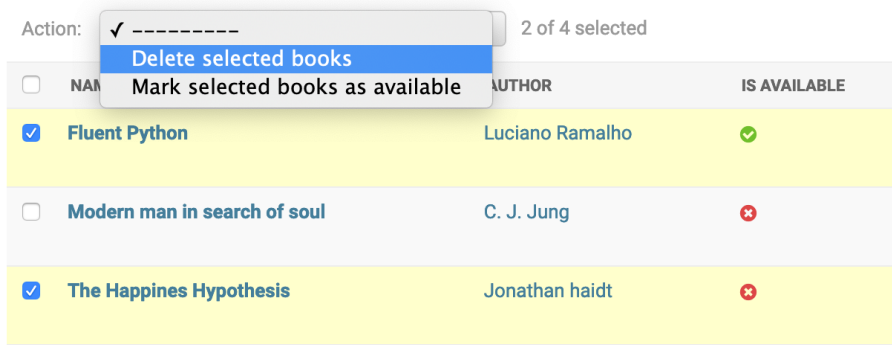
When a model is heavily used to update the content, it makes to sense to allow bulk edits on the models.

```
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.

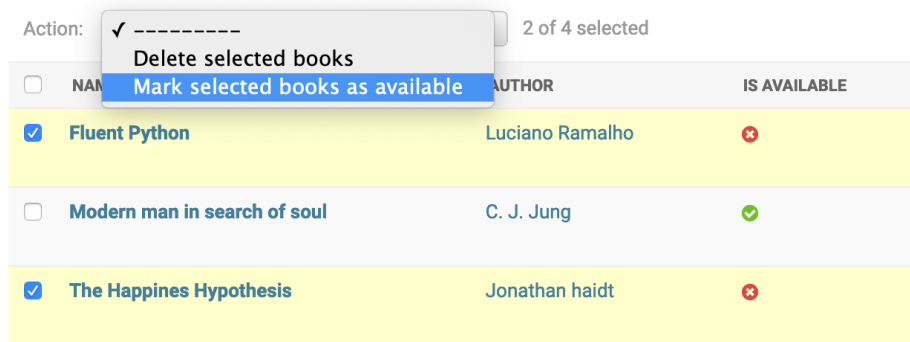
In our books admin, we can select a bunch of books and delete them.



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"
```



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.

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.

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

ORM GOTCHAS

N+1 Queries

8.1 Caching

8.2 Eager evaluation

8.3 Lazy evaluation

`book.author.id` `book.author_id`

`qs.exists()`

`.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)
```

DASHBOARDS

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

SECURING DJANGO ADMIN

10.1 Securing Server

10.1.1 VPN

10.1.2 FIREWALL

80/443

10.1.3 https

10.2 Securing Django

`python manage.py check --deploy`

10.2.1 allowed hosts

10.2.2 Disable DEBUG

10.2.3 Change default url

10.2.4 Ensuring proper ACL

10.2.5 Honeypot

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

10.2.6 2FA

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

10.2.7 ENVironment

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

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

CHAPTER ELEVEN

FINAL WORDS

Think about workflows.

Don't waster too much time.