Admin Site Summary

> Setting up the Admin Site
>
> - accessible at `/admin`
> - we need a super user to access (`python manage.py createsuperuser`)
> - session app is required for admin app to work
> - Groups and Users are there by default (Stored in the **auth** app)
> - Reset password for superuser (`python manage.py changepassword 'user'`)
>
> ## Change the site title
>
> ```
> <<urls module>>
>
> admin.site.site_header = 'custom header'
> ```
>
> ## Change index header
>
> ```
> <<urls module>>
>
> admin.site.index_title = 'custom title'
> ```

> Registering Models
>
> - Every Django app has the `admin` module which is intended for customizing the admin interface
>
> in the admin module of `store` app
>
> ```
> from . import models
>
> admin.site.register(models.Collection)
> ```

Added functionality after registering a model

- Model list view
- Adding new instances
- Changing existing instances

Model representation

By default we see something like `Collection object(n)` and we need to change it. How?

```
class Collection:
        def __str__(self) -> str:
                return self.title
```

Sorting instances

```
class Collection:
        class Meta:
                ordering = ['title'] # list of field(s)
```

Customizing the List Page

Create admin model

```
@admin.register(models.Product)
class ProductAdmin(admin.ModelAdmin):
        list_display = ['title', 'unit_price']
```

Editable fields from the list page

```
list_editable = ['unit_price']
```

Pagination

```
list_per_page = 10
```

> Search for `Django Model Admin` for more options

> Adding Computed Columns
>
> - **Inventory Status** for products (less than 10)

```
@admin.display(ordering='inventory')
def inventory_status(self, product):
        if product.inventory < 10:
                return 'Low'
        else:
                return 'Ok'
```

> Selecting Related Objects
>
> - Show the collection of each product

```
def collection_title(self, product):
        return product.collection.title
```

This approach sends a lot of queries so:

```
class ProductAdmin:
        list_select_related = ['collection']
```

> Overriding the Base QuerySet
>
> Sometimes we need to override the QuerySet used for rendering the list page.
>
> - Show the number of products for each collection

```
@admin.register(models.Collection)
class CollectionAdmin(model.ModelAdmin):
        list_display = ['title', 'products_count']
```

```
        @admin.display(ordering='products_count')
        def products_count(self, collection):
                return collection.product_count
        def get_queryset(self, request):
                return
Collection.objects.select_related('product')
                .annotate(
                        products_count=Count('product')
                )
```

Providing Links to other Pages

- Click on `products_count` and see the all products of that collection
- Instead of a number we need to return a string containing an HTML link
- For generating HTML links we must import a utility function

```
from django.utils.html import fromat_html, urlencode
from django.urls import reverse

@admin.display(ordering='products_count')
def products_count(self, collection):
    url = (
                reverse('admin:store_product_changelist')
                + '?'
                + urlencode({'collection__id':
str(collection.id)})
    )

    return format_html(
        '<a href="{}">{}</a>',
        url,
        collection.products_count
    )

def get_queryset(self, request):
    return Collection.objects.annotate(
```

```
            products_count=Count('product')
        )
```

1. Getting the products count (`get_queryset`)
2. Generate the URL (`reverse` , `urlencode()`)
3. Return the count as a link (`fromat_html`)

Adding Search to List Page

- add searching to customers page
- user `search_fields`
- default pattern (`%%` , `icontains`)
- preferred pattern (`^` , `istartswith`)

```
search_fields = [
            'first_name__istartswith',
            'last_name__istartswith'
]
search_help_text = 'Search using First or Last Name'
```

Adding Filtering to the List Page

- Filter Products by **Collection** and **Last Update**

```
class ProductAdmin(admin.ModelAdmin):
            list_filter = ['collection', 'last_update']
```

Creating Custom Filters

- See the products with low inventory

```
# in the admin module
class InventoryFilter(admin.SimpleListFilter):
        title = 'inventory'
        parameter_name = 'inventory'

        # Required Methods
        def lookups(self, request, model_admin):
                return [
                        ('<10', 'Low')
                ]
        def queryset(self, request, queryset):
                # returns the selected filter
                if self.value() == '<10' # BP: set to a
variable
                        return
queryset.filter(inventory__lt=10)
```

- `title` - self explanatory
- `parameter_name` - this is the parameter used in the query string in the URL
- `lookups()` - what items should appear below the filter title
    - Returns a list of tuples: (`Actual Value`, `human readable description`)
- `queryset` - this is where we implement the filtering logic

Now to use it

```
class ProductAdmin(admin.ModelAdmin):
        list_filter = ['collection', 'last_update',
InventoryFilter]
```

Creating Custom Actions

- Sometimes we want to apply a custom action to one or many items in a list page
- By default, all list pages have the `Delete selected items` action

Now let's define custom ones:

- Clearing the inventory of one or more products in one go

```
# in the admin module
class ProductAdmin(admin.ModelAdmin):
        @admin.action(description='Clear inventory')
        def clear_inventory(self, request, queryset):
                updated_count =
queryset.update(inventory=0)
                self.message_user(
                        request,
                        f'{updated_count} products were
updated successfully.'
                )
```

- `queryset` in parameters, indicates the items that a user has selected
- `queryset.update()` will immediately update the database and return the number of updated records.
- `self.message_user()` - every `ModelAdmin` contains this method for showing a message to the user.
    - first argument - *Request*
    - second argument - *The message*
    - third argument - *Level*

```
class ProductAdmin(admin.ModelAdmin):
        actions = ['clear_inventory']
```

Customizing Forms

- Changing the form for **adding** and **updating** models.
- These forms are generated based on the definition of a **Model**

```
fields = ['fields',]
```

OR

```
exclude = ['fields']
```

we also have

```
readonly_fields = ['fields']
```

Auto-population

- This scenario is about the `slug` field

```
prepopulated_fields = {
        'slug': ['title'] # can pass multiple fields
}
```

Auto-Complete field

- For Drop-down lists that have lots of items
    - Results in better performance

```
autocomplete_fields = ['collection']
```

> Tip
>
> You may need to define `search_fields` for the other end of the relationship in order for `autocomplete_fields` to work.

Adding Data Validation

- By Default our forms provide basic data validation logic

> Price can be set to 0 which is logically wrong

```
from django.core.validators import MinValueValidator
# See more validators → Search Django Validator

class Product(models.Model):
        unit_price = DecimalField(
                max_digits=6,
                decimal_places=2,
                validators=[MinValueValidator(1)] # Supply
more if needed
        )   # we can supply a custom message as well
```

- Null - Database Level
- Blank - Form level

Editing Children using Inlines

- Order Item editing via Order

```
# Admin Module
class OrderItemInline(admin.TabularInline):
        model = OrderItem
        # optional
        extra = 0 # Default is 3
        min_num = 1 # If less → validation error
        max_num = 10

class OrderAdmin(admin.ModelAdmin):
        inlines = [OrderItemInline]
```

- **Tabular Inline** is a table of rows and columns
- **Stacked Inline** is a form

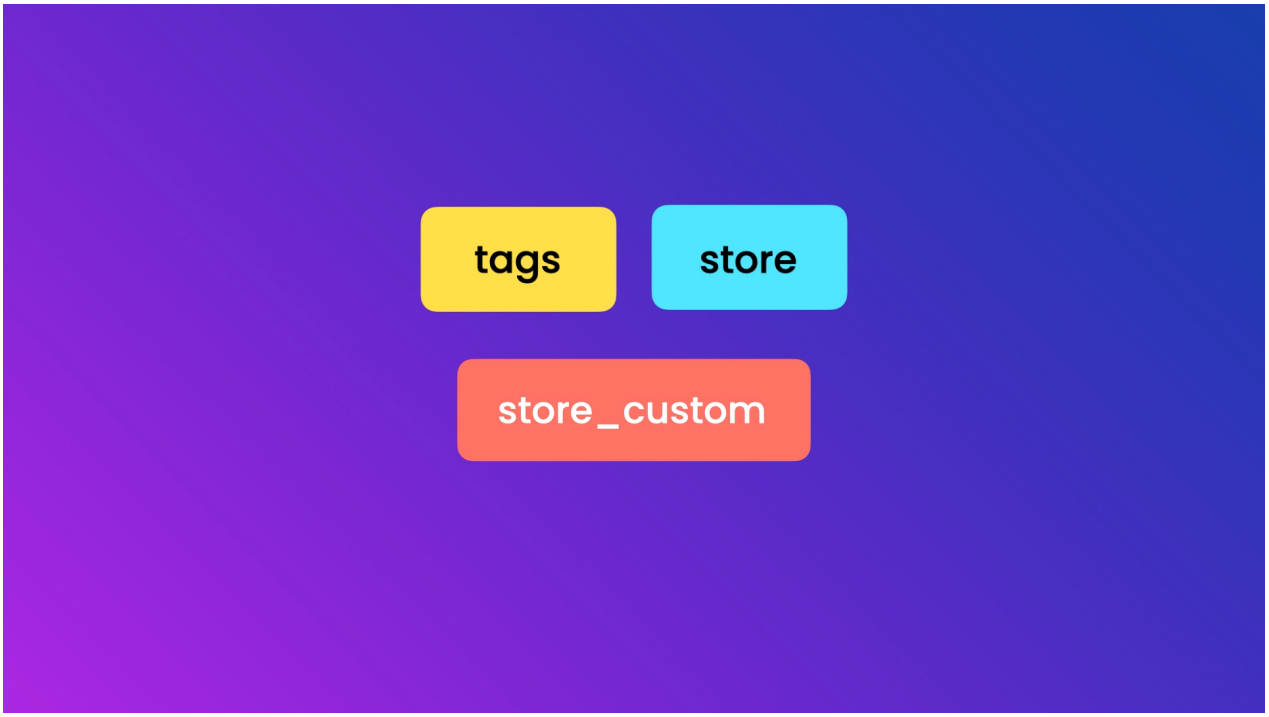
Using Generic Relations

- In the products form, we want to add a section for managing tags
  - register `tag` model
  - create an inline for tags in products form
    - `GenericTabularInline`
    - `GenericStackedInline`
  - Pro Tip: convert dropdown to `autocomplete_field`
    - define search fields for this

> ⚠ Warning
>
> This Approach couples our apps together which ruins the self contained rule

Solution: Extending Pluggable Apps

Extending Pluggable Apps



- **Tags** and **Store** are going to self-contained

- **Store Custom** / **Core** is specific to this project

Logic

- Combine features of these two apps in **Store Custom**
- Extend `ProductAdmin`
- Unregister old `ProductAdmin`
- Register new `ProductAdmin`
