

**Introduction**:

Now We will be setting up an admin interface for managing our data. We will talk about

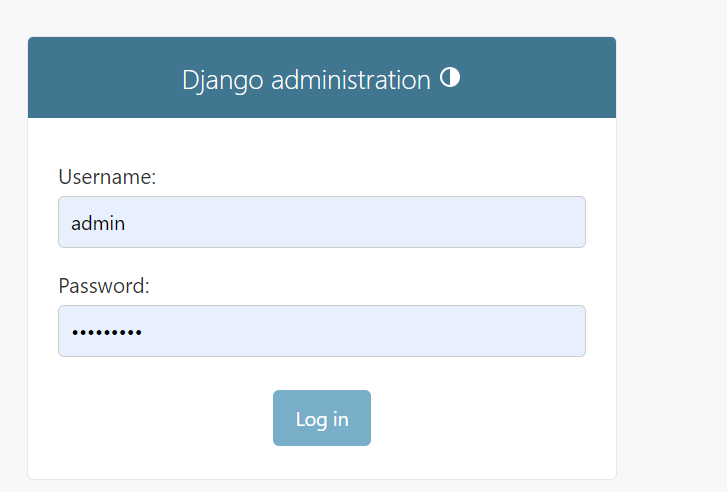
1. Various ways to customize the admin interface.
2. Adding computed columns.
3. Loading related objects.
4. Add searching and filtering.
5. Implement custom actions.
6. Add Data validations

And much more…

By the end of this section , we will have full-fledged admin interface for managing our data.

**Setting up the admin site**:

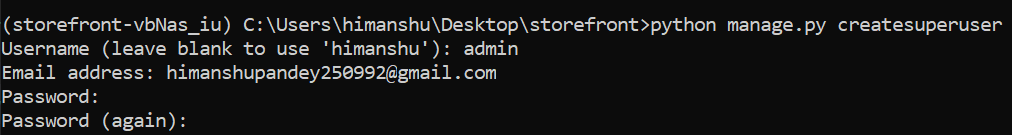
Every Django app comes with an admin interface, which we can access at <http://127.0.0.1:8000/admin>

🡨login screen

🡪To create a new user run this command,

python manage.py createsuperuser

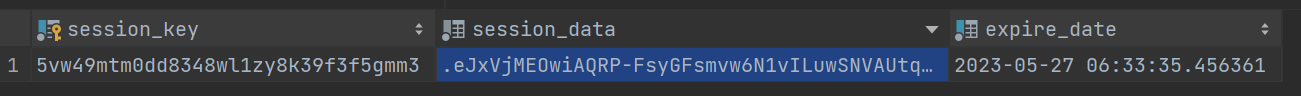
Fill these fields



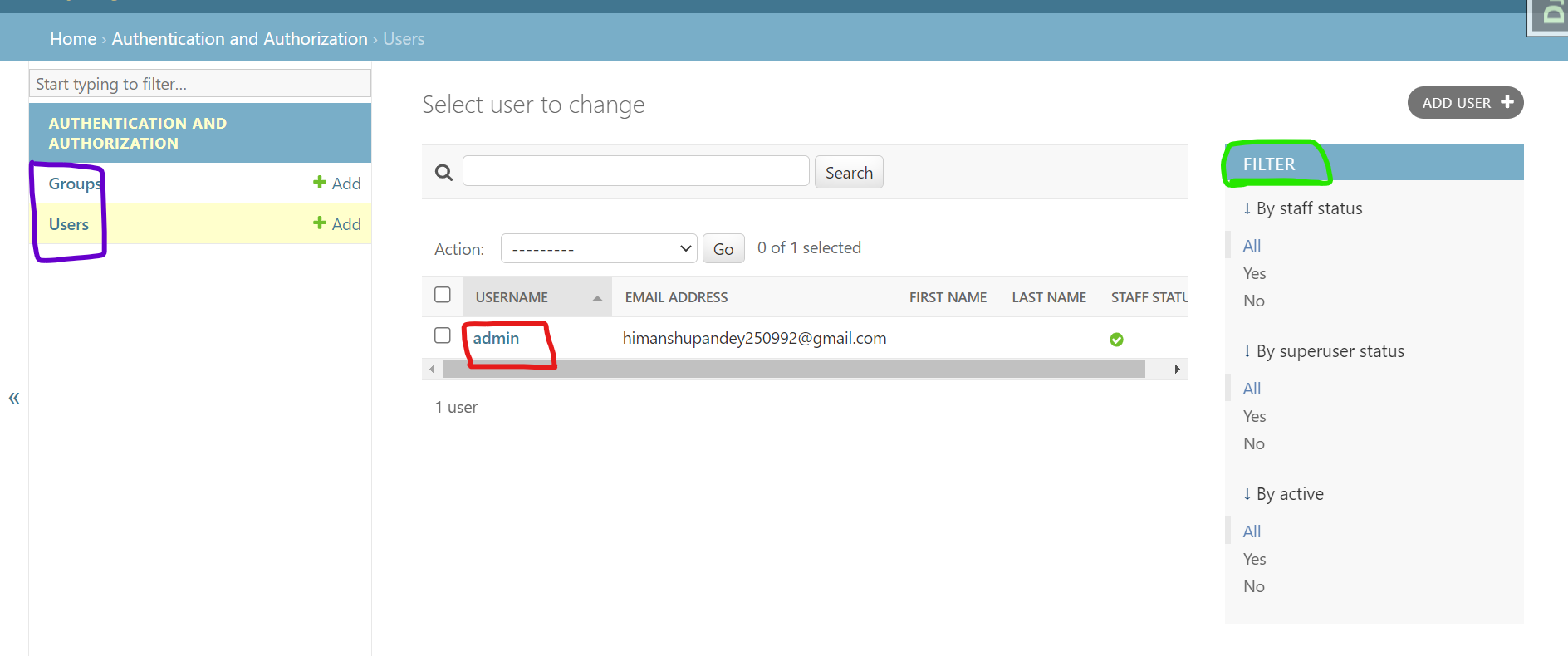
Finally we get message

*Superuser created successfully*.

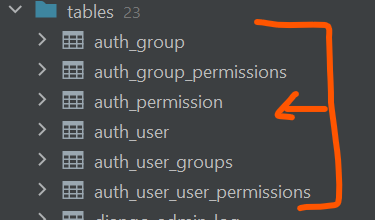
Note: When we log into admin panel, a session record is created automatically in sessions table.



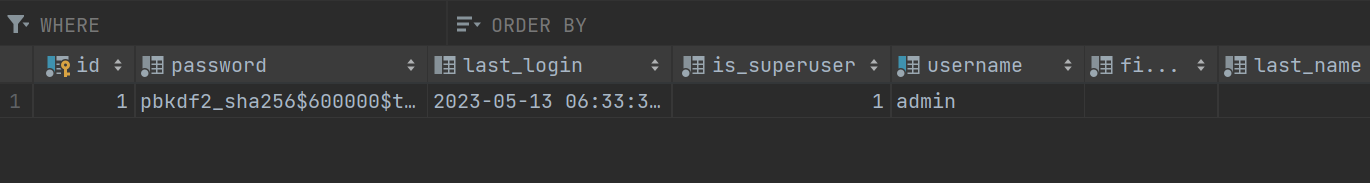
Back to our admin panel



We see no list of groups and just a single user ‘admin’. These users and groups are stored in tables of auth app.

🡨auth app has a bunch of tables for managing *groups*, *permissions* and *users*.

In the *user* table we see a long row of user related fields



Password is in encrypted format. But we can see last login date and time, Superuser or not, email.

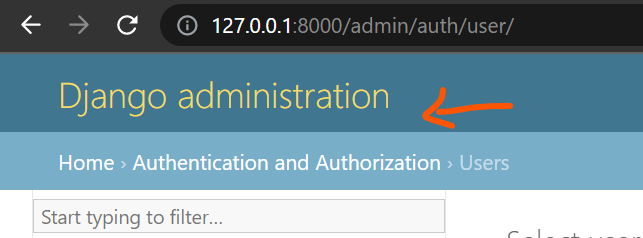
Note: If *is\_staff* value is True then it means this is an admin user.

*To reset the password*:

python manage.py changepassword admin

*To change the name of admin panel heading*:

Let us try to change the name here



From Django administration to something meaningful

1. Go to storefront project and navigate to its urls.py module. Here all the URLs of the admin app are hooked up this URL

urlpatterns = [

    path("admin/", admin.site.urls),

    path("playground/", include("playground.urls")),

    path("\_\_debug\_\_/", include("debug\_toolbar.urls")),

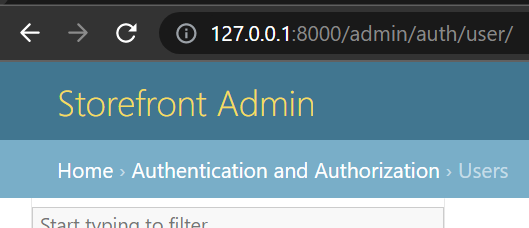
]

So we can see here that any URL starting from admin/ is going to be routed to admin app.

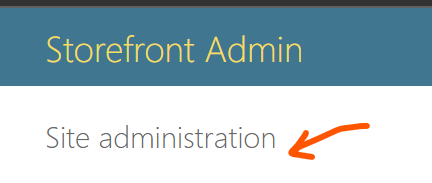
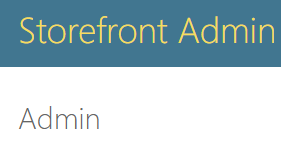
1. Since we already have admin app imported here, it is a good place to add some customizations for admin panel.

from django.contrib import admin

admin.site.site\_header = 'Storefront Admin'

🡨We changed *header* on the top using site\_header.

1. We can also change this *title* here from Site administration to Admin.

 🡪🡪🡪

By using setting index\_title.

admin.site.index\_title = "Admin"

**Registering Models**:

Now we will see, how we can register our models so that we can manage them from our admin site.

So back to our *store* app, we can see a admin.py module.

*Here we write all the code for customizing the administration panel for store app*.

We already have admin module imported at the top.

from django.contrib import admin

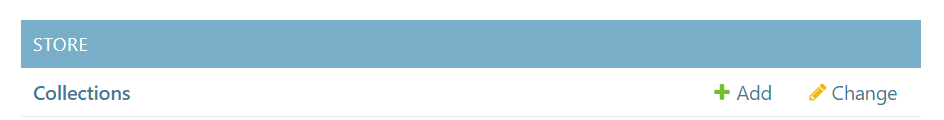
Next we use admin.site.register method and provide it with model that we want to register in our admin. So do another import of models module from the current directory as well.

from django.contrib import admin

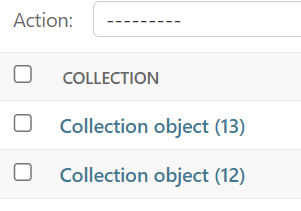
from . import models

admin.site.register(models.Collection)

Here we have registered our Collections model. In our admin panel we can see our *Store* app with *Collections*.

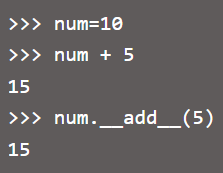


Inside Collection table we see default representation of a model object(*each object is a record in collection table*)

🡨Instead of this, we should show title of each collection.

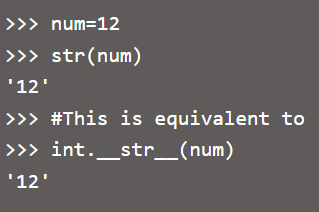
A little detour: **Magic methods** in python are the special methods that start and end with double underscores. *They are not meant to be invoked directly by us, but the invocation happens internally from the class on a certain action*.

For example, when we add two numbers using + operator, internally the \_\_add\_\_() method will be called



\_\_str\_\_() method

This method *is overridden to return a printable string representation of any user defined class*.



--------------------------------End of detour------------------------------

Now, How can we change string representation of an object in python. We override the magic \_\_str\_\_ method.

Every Python object has a magic method that is called when that object is converted to a string.

In order to do that, go to *Collection* *model* and create a function:

    def \_\_str\_\_(self) -> str:

        return super().\_\_str\_\_()

Note: **->** is the type annotation, which basically says that \_\_str\_\_(self) returns a string object.

Inside we have , super().\_\_str\_\_(), a default representation which returns a string like this…



We are now going to change the default implementation by returning *self.title*.

class Collection(models.Model):

    # collection\_id field created by Django

    title = models.CharField(max\_length=255)

    featured\_product = models.ForeignKey(

        "Product", on\_delete=models.SET\_NULL, null=True, related\_name="+"

    )

    def \_\_str\_\_(self) -> str:

        return self.title

🡨Now, we can see all the collections with their title.

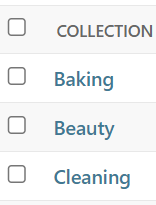
*Sorting the list of collections*:

Inside the Collection model, we can define a Meta class for specifying the default ordering of collection objects.

    class Meta:

        ordering = ["title"]

Here we use ordering *meta option* and set it to a list of items sorted by their title in ascending order.

🡨All collections are sorted now.

**Exercise**:

Register the Product model using the same techniques.

1. Go to admin module of store app and register Product model.

admin.site.register(models.Product)

1. Go to Product class and provide title to represent Product objects and then sort them using ordering meta option

class Product(models.Model):

    title = models.CharField(max\_length=255)

    slug = models.SlugField()

    description = models.TextField()

    unit\_price = models.DecimalField(max\_digits=6, decimal\_places=2)

    inventory = models.IntegerField()

    last\_update = models.DateTimeField(auto\_now=True)

    collection = models.ForeignKey(Collection, on\_delete=models.PROTECT)

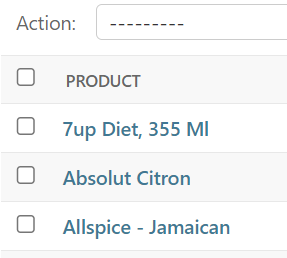
    promotions = models.ManyToManyField(Promotion)

    def \_\_str\_\_(self) -> str:

        return self.title

    class Meta:

        ordering = ["title"]

🡨 Result

**Customizing the List page**:

Now, we have a list of Collection and Product objects in our panel in the form of list, Next we will work on customizing that list page.

We will see

🡪How to add new columns in our list.

🡪How to make them editable.

🡪Change number of items and so on…

Back to our admin.py module, We create a new class called *ProductAdmin*(*we can name it anything but by convention we use name of our model followed by the word Admin*).

Note: With this class we can specify how we want to view or edit our products therefore, we inherit from admin.ModelAdmin class.

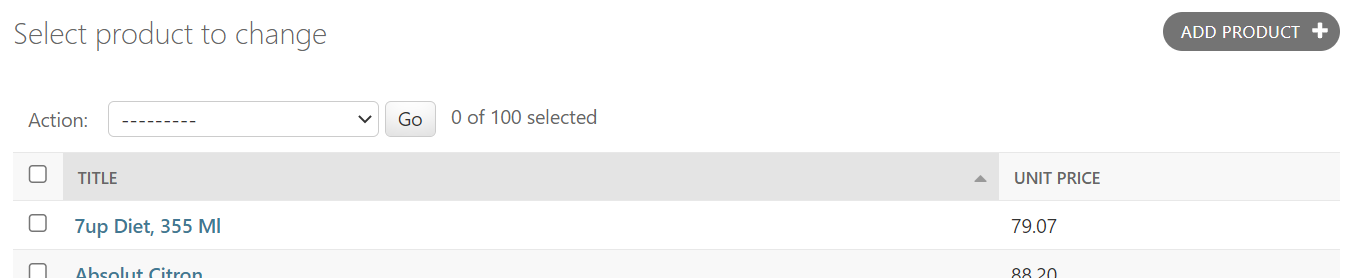
Inside our ProductAdmin class we can set a bunch of attributes to customize the list page. One of these attributes is list\_display for specifying the list of fields we want to display(e.g. title and unit price ).

class ProductAdmin(admin.ModelAdmin):

    list\_display = ["title", "unit\_price"]

Now we have a new class, we need to pass it while registering Product model.

admin.site.register(models.Product, ProductAdmin)



This is one way to do this, but there is a *shortcut* as well:

Note: We can use *register decorato*r over our ProductAdmin class.

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    list\_display = ["title", "unit\_price"]

This way we are saying ProductAdmin is the admin model for the Product class. Due to this decorator we do not need admin.site.register() method at the end.

Another attribute we can set is list\_editable and with this we can specify the fields that can be edited on the list page.

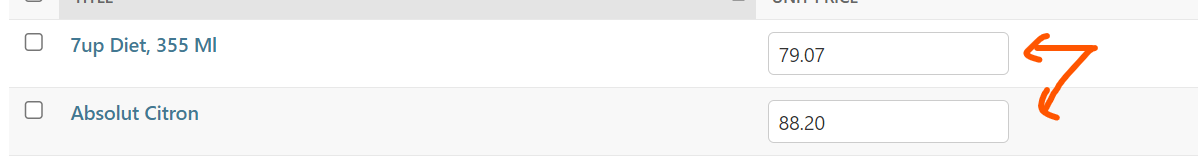
@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    list\_display = ["title", "unit\_price"]

    list\_editable = ["unit\_price"]

now, the unit price field is editable and we can change price of multiple products in one go.



We can also set list\_per\_page for setting pagination(number of objects per page)

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    list\_display = ["title", "unit\_price"]

    list\_editable = ["unit\_price"]

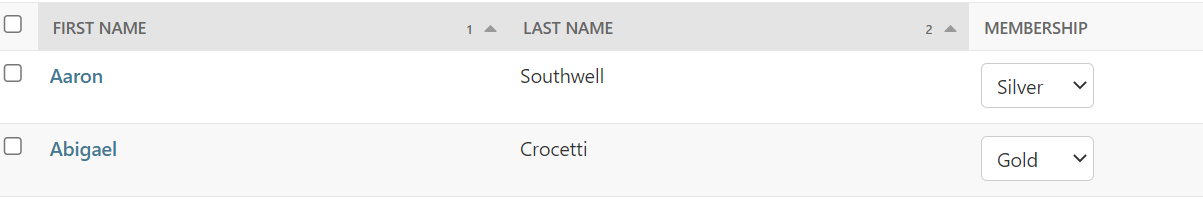
    list\_per\_page = 10

Now we will see 10 products per page.

Note: If you want to see complete list of options we can use here to customize admin page google, *Django Model Admin* 🡪 *Model Admin options*.

**Exercise**:

Create a page for Customer table as well in this way



Solution:

@admin.register(models.Customer)

class CustomerAdmin(admin.ModelAdmin):

    list\_display = ["first\_name", "last\_name", "membership"]

    list\_editable = ["membership"]

    ordering = ["first\_name", "last\_name"]

    list\_per\_page = 10

**Adding computed columns**:

Let us see how can we add a computed column to the list of products.

**Example**, We are adding a new column called *inventory status* which will show ‘*low*’ if inventory of a product is less than 10.

Solution:

Here we define a new function in our ProductAdmin class called inventory\_status with two parameters called ***self*** and ***product***,

def inventory\_status(self, product):

Now depending on inventory of product, we will return an inventory status.

    def inventory\_status(self, product):

        if product.inventory < 10:

            return "Low"

        return "Ok"

In list\_display we can add name of our function,

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    list\_display = ["title", "unit\_price", "inventory\_status"]

    list\_editable = ["unit\_price"]

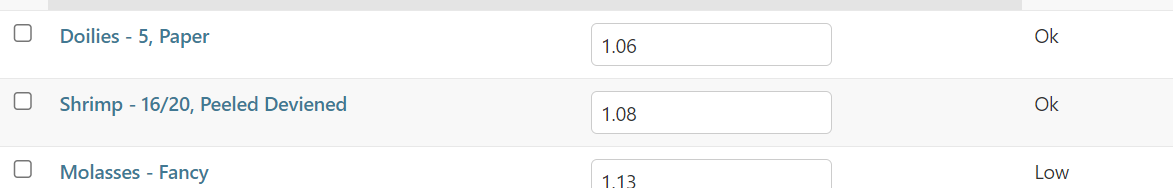
    list\_per\_page = 20

    def inventory\_status(self, product):

        if product.inventory < 10:

            return "Low"

        return "Ok"



But there is one small issue, This inventory\_status column is not sortable by default because Django does not know how to sort the content of this column.

We can use admin.display decorator method to apply sorting,

@admin.display()

Here we specify field that should be used for sorting data in inventory\_status colum.

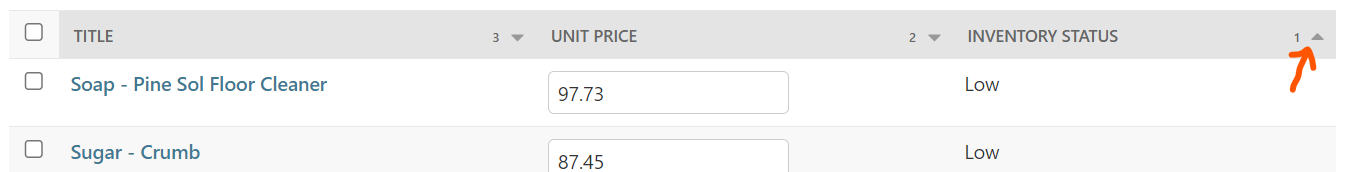
    @admin.display(ordering="inventory")

    def inventory\_status(self, product):

        if product.inventory < 10:

            return "Low"

        return "Ok"



Now the sorting is applied on this column and we can sort in ascending or descending order.

**Selecting related objects**:

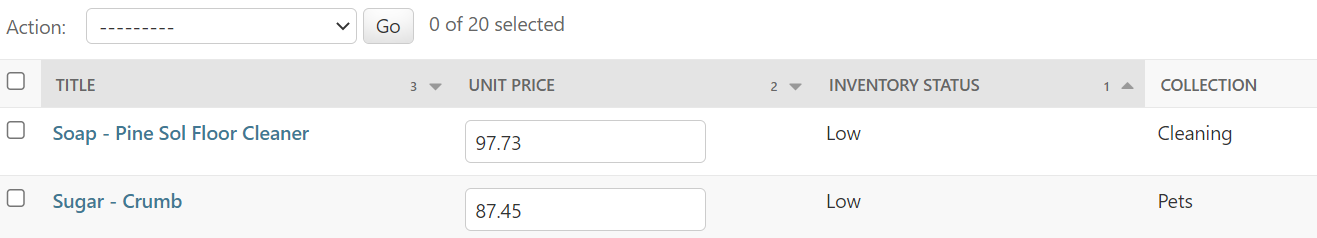
Let us talk about loading related objects. Example, In the list of products we need to see collection related to each product.

So let us add a new field in the list\_display called *collection*.

class ProductAdmin(admin.ModelAdmin):

    list\_display = ["title", "unit\_price", "inventory\_status", "collection"]

and we can see our list of products populated with this.



Note: If did not override the \_\_str\_\_ method in Collection class earlier, we would not be able to see titles of each collection. Therefore when showing the product Django will call this method to get the title of the collection for that product.

Now let us open Django debug toolbar to see queries in SQL,



Here we see all column from Product table, followed by all columns from collection table and Inner join between them

Pretty straightforward so far…

*Showing a particular field from column*:

What if we do not want to show string representation of collection table here and instead want to show some other field. *Unfortunately we cannot use this syntax, collection\_\_ followed by name of field which we want to reference*.

Solution 1:

🡪Define a function called collection\_title inside ProductAdmin class with two parameters as *self* and *product*, because we are rendering a bunch of products.

    def collection\_title(self, product):

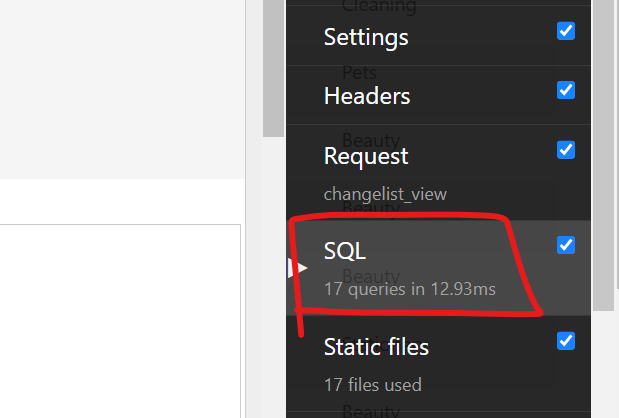
🡪return product.collection.title

    def collection\_title(self, product):

        return product.collection.title

🡪Add this function in list\_display attribute.

list\_display = ["title", "unit\_price", "inventory\_status", "collection\_title"]

🡨We have 17 queries, 10 extra because for each product Django is sending an extra query to the database to read collection of that product.

Solution 2: (*better way*)

In the previous sections we learned about query set where we have a method called select\_related to preload the related field.

Inside ModelAdmin we have a similar attribute for the same purpose called list\_select\_related which we set to list of fields

we want to preload.

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    list\_display = ["title", "unit\_price", "inventory\_status", "collection\_title"]

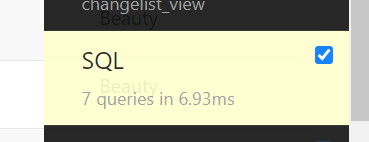
    list\_editable = ["unit\_price"]

    list\_per\_page = 10

    list\_select\_related = ["collection"] 🡨 change here

    def collection\_title(self, product):

        return product.collection.title

🡨 Now we are back to 7 queries ☺

**Overriding the base query set**:

Sometimes we need to override the base query set used for rendering a list page.

For example, In the list of collections I want to add a new field called products\_count to count the number of product in each collection.

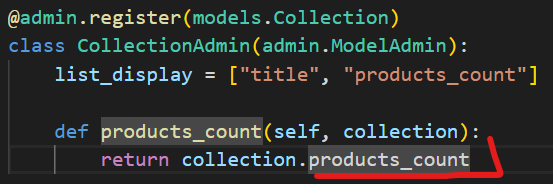
@admin.register(models.Collection)

class CollectionAdmin(admin.ModelAdmin):

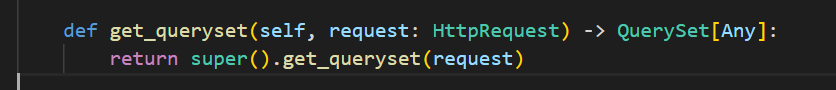
    list\_display = ["title", 'products\_count']

But the problem is we do not have a field with name products\_count. So we have to treat this as a *computed* field.

So we define a new method with this name,

🡨since we do not have this field, we need to override the query set on this page and annotate our collection with a number of product.

*get\_queryset method*:



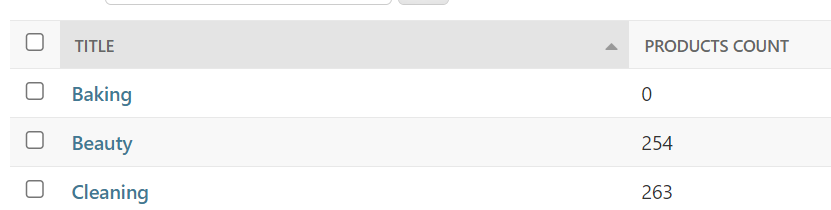
Every model admin has a get\_queryset method which we can override. (*we can remove type annotations like HttpRequest*…)

As you can see we are returning a queryset from this method, but we do not want to return it as it is.

So we chain annotate method where we give the count of products in each collection using Count object.

def get\_queryset(self, request):

  return super().get\_queryset(request).annotate(products\_count=Count("product"))



But this column is not sortable because Django does not know which field to use to sort this column. So to solve this problem, we need to decorate our products\_count method with admin.display() and set the ordering to products\_count

  @admin.display(ordering="products\_count")

  def products\_count(self, collection):

    return collection.products\_count

def get\_queryset(self, request):

  return super().get\_queryset(request).annotate(products\_count=Count("product"))

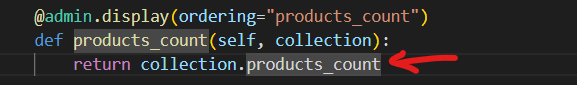


**Providing links to other pages**:

Let us see, how we can add links with list of our collections.

So that when we click on these links we can see products in each collection.

Now back to our *products\_count* method *instead of returning a number we should return a string containing an HTML link*.

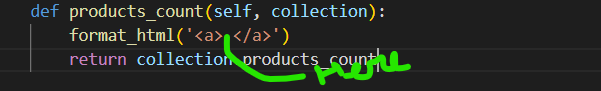


To generate an HTML link, we should import a utility function format\_html on the top.

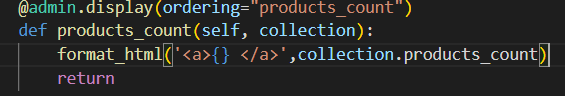
from django.utils.html import format\_html

In our products\_count method, we call format\_html method and give it a format string(*some html code*).

In html we can refer to a link using anchor <a> </a>tag and in between these tags we can type the link text.



*Here*, we type pair of braces {} where we will pass number of products. So we pass collection.products\_count as the second argument of format\_html (🡪need to study)function.



For the actual link, we will give anchor tag, a href attribute. For now let us just use <http://google.com> as the URL.

    @admin.display(ordering="products\_count")

    def products\_count(self, collection):

        return format\_html(

            '<a href="http://google.com">{}</a>', collection.products\_count

        )

As per Implementation up to this point, we have clickable links in our page under products\_count column and when we click on them we are redirected to google.

*Sending users to product page*:

On the products page observe the URL,

<http://127.0.0.1:8000/admin/store/product/>

admin🡪store🡪product.

We do not want to hardcode this URL in our code because it can potentially change in the future. So we should *ask Django to give us the URL for this page*.

To do that, we need to import another utility function called reverse.

from django.urls import reverse

We give this function a special argument,

reverse(admin: app\_model\_page)

admin: name of app \_ name of model \_ name of page

In our case it will be

reverse('admin:store\_product\_changelist')

Note: Django admin’s “change list” is the page that lists all objects of a given model.

    @admin.display(ordering="products\_count")

    def products\_count(self, collection):

        url = reverse("admin:store\_product\_changelist")

        return format\_html('<a href="{}">{}</a>', url, collection.products\_count)

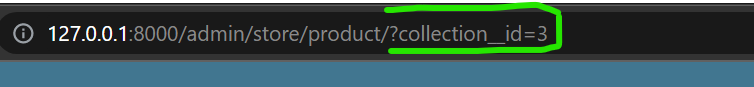
Here we call reverse function to get the url and replace google URL with it.

With this implementation, when we click on the link we are directed to admin/store/product change list page. But there is no filter applied here.

*By filter we mean that we click on beauty collection only those products that are under that collection should show*. Right now we are seeing all the products irrespective of the collection id.

*Applying filter in the URL*:

To apply a filter, we need to append a query string in the URL

 like this.

First add a ? with the current URL

url = reverse("admin:store\_product\_changelist") + "?"

Note: When we add a **?** in the URL, it indicates beginning of a query string.

Next we need to add actual query part followed by **?** dynamically. So we use another utility function called urlencode.

from django.utils.html import format\_html, urlencode

We will call this function and give it a dictionary because a query string can contain multiple key value pairs.

        url = (

            reverse("admin:store\_product\_changelist")

            + "?"

            + urlencode({"collection\_\_id": str(collection.id)})

        )

Note: We are using str() to convert collection.id which is a number into string. *Side note(I checked without str method it is still working fine).*

With this implementation when we click on any collection we only see products inside a specific collection!

*Complete collection admin model implementation up to this point is below*:

@admin.register(models.Collection)

class CollectionAdmin(admin.ModelAdmin):

    list\_display = ["title", "products\_count"]

    @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 super().get\_queryset(request).annotate(products\_count=Count("product"))

**Exercise**: Go to customers page, add a new column for viewing orders of each customer.

@admin.register(models.Customer)

class CustomerAdmin(admin.ModelAdmin):

    list\_display = ["first\_name", "last\_name", "membership", "orders\_count"]

    list\_editable = ["membership"]

    list\_per\_page = 10

    ordering = ["first\_name", "last\_name"]

    @admin.display(ordering="orders\_count")

    def orders\_count(self, customer):

        url = (

            reverse("admin:store\_order\_changelist")

            + "?"

            + urlencode({"customer\_\_id": str(customer.id)})

        )

        return format\_html('<a href="{}">{}</a>', url, customer.orders\_count)

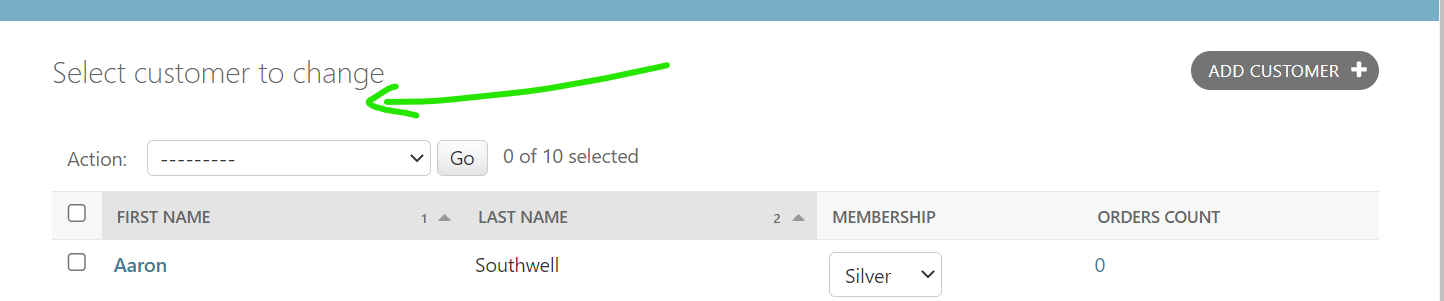
    def get\_queryset(self, request):

        return super().get\_queryset(request).annotate(orders\_count=Count("order"))

**Adding search to the list page:**

**Exercise**: Add search fields in the customer page .

Right now there is no search bar on our customer’s page



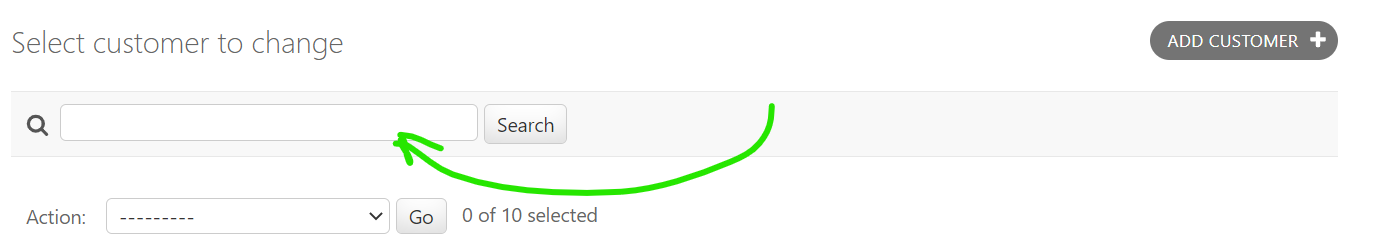
So to solve this,

In the customer admin class we will set search\_fields attribute to the list of fields we want to use for searching.

search\_fields = ["first\_name", "last\_name"]

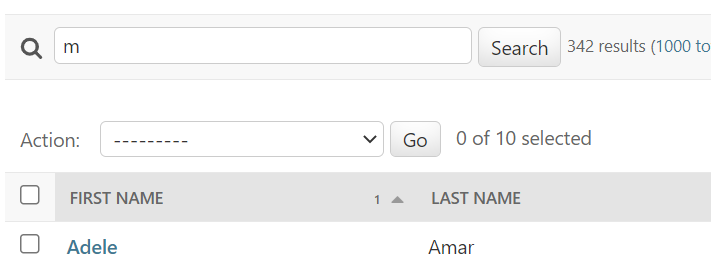
Useful tip: When you have multiple attributes, Always sort all attributes inside model / class in an alphabetical order to make the code more readable.

After adding search\_fields, we can see a beautiful search bar appear in the customer page.



So if search for ‘cat’, we see all customers with cat in their first of last name,

Now, if we search ‘m’ in the search bar , we see the issue



We get ‘Adele Amar’ , technically it does not make sense, we expect to see people whose first or last name begin with ‘m’.

So back to the search\_fields, we need to use one of the lookup types called \_\_startswith.

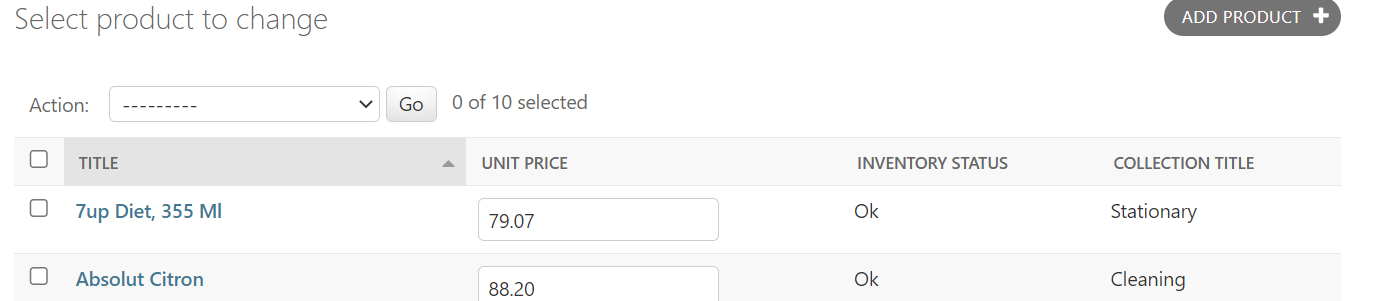
search\_fields = ["first\_name\_\_startswith", "last\_name\_\_startswith"]

Note: to make lookup types case insensitive add an ‘**i**’ before it.

search\_fields = ["first\_name\_\_istartswith", "last\_name\_\_istartswith"]

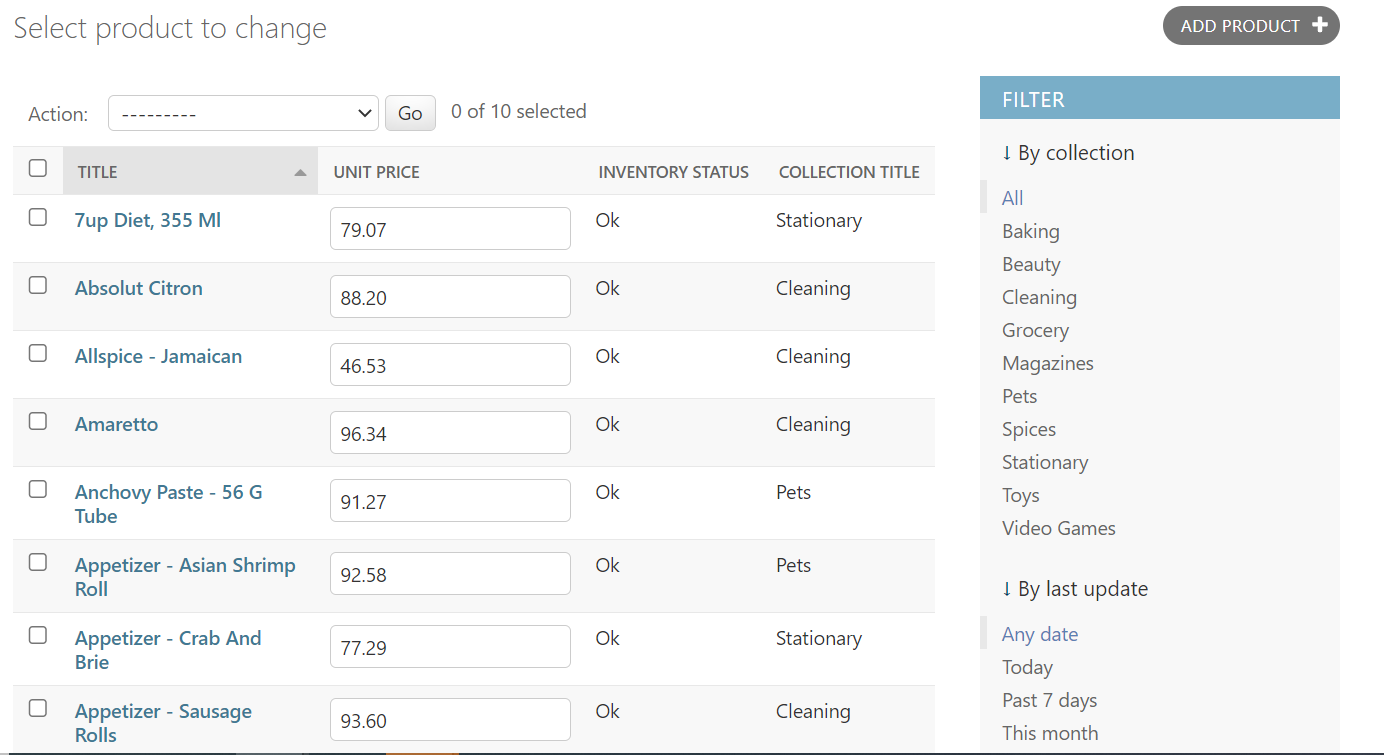
**Adding filtering to the list page:**

**Exercise**: Add filtering on the products page. Filter our product by their collection and last update.



Add a new attribute in the ProductAdmin class called list\_filter and set it to a list of collection and last\_update field.

list\_filter = ["collection", "last\_update"]



We have a very useful Filter section , just by writing a single line of code.

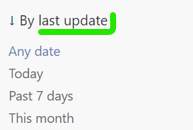
*Adding a custom filter*:

Suppose we only want to see products with low inventory.

For this we will create a new class *InventoryFilter*, which should extend to admin.SimpleListFilter.

class InventoryFilter(admin.SimpleListFilter):

Inside it we should set a couple of attributes, one of them is title, which will appear after *By* , for now set it to ‘*inventory’*



Second attribute is parameter\_name, which will be used in query string.

Note: When we click on any collection name inside collection filter, we see a query string in the end of URL

[http://127.0.0.1:8000/admin/store/product/?**collection\_\_id\_\_exact=3**](http://127.0.0.1:8000/admin/store/product/?collection__id__exact=3). This part **collection\_\_id** is set by parameter\_name.

class InventoryFilter(admin.SimpleListFilter):

    title = "inventory"

    parameter\_name = "inventory"

Now we have two methods that we need to implement here,

One of them is lookups, with this method we can specify what items should appear inside *inventory* filter.

--------------------------------------------------------------------------------

Tip: when we type methods like lookups or queryset , we see default type annotations, which we can keep or remove

    def lookups(self, request: Any, model\_admin: Any) -> List[Tuple[Any, str]]:

        return super().lookups(request, model\_admin)

after removing type annotations,

    def lookups(self, request, model\_admin):

        return super().lookups(request, model\_admin)

--------------------------------------------------------------------------------

Inside lookups function, we are going to return list of tuples. Each tuple represent sub-filters inside inventory filter.

    def lookups(self, request, model\_admin):

        return [('<10','low')]

*These tuples has two values inside them first one is the actual value that we are using for filtering and values is the human readable description*.

Second method is the queryset, this is where we implement filtering logic like this,

    def queryset(self, request, queryset):

        if self.value() == "<10":

            queryset.filter(inventory\_\_lt=10)

self.value() returns the selected filter. If this is equal to *‘<10’* that we defined in lookups method. Then we will take the queryset and apply filter method on it.

class InventoryFilter(admin.SimpleListFilter):

    title = "inventory"

    parameter\_name = "inventory"

    def lookups(self, request, model\_admin):

        return [("<10", "low")]

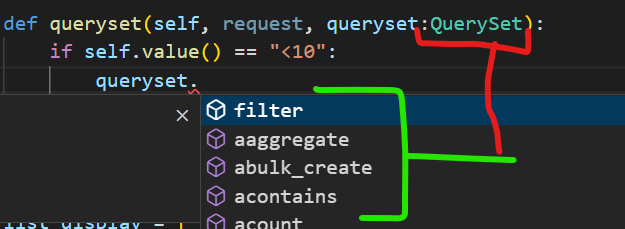
    def queryset(self, request, queryset: QuerySet):

        if self.value() == "<10":

          return queryset.filter(inventory\_\_lt=10)

*inventory\_\_lt=10* will return products with inventory less than 10.

Note: If we decorate our parameters in queryset method with type annotations, intellisense will work to show all the related methods. This is benefit of using type annotations.



To use this filter, simply call the name of the class inside list\_filter of ProductAdmin class

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    list\_display = ["title", "unit\_price", "inventory\_status", "collection\_title"]

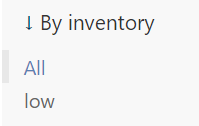
    list\_editable = ["unit\_price"]

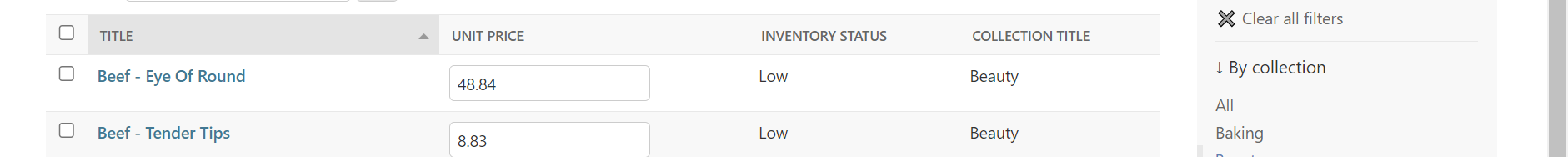
    list\_per\_page = 10

    list\_select\_related = ["collection"]

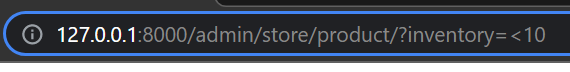
    list\_filter = ["collection", "last\_update", InventoryFilter]

We can see our new filter now

 After clicking on ‘low’ , we only see the products with Low inventory.

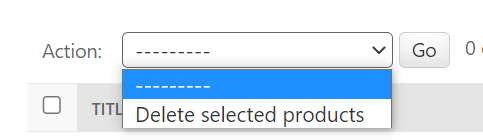


URL



**Creating custom actions**:

Let us talk about defining custom actions. By default, each page comes with a delete action. For deleting multiple objects in one go.



We can extend this list and register our own custom actions.

For example, we want to define a custom action for clearing the inventory of a bunch of products in one go and thus setting the inventory to zero.

In the ProductAdmin class, we will define a new method called *clear\_inventory*. Here we need three parameters self, request(*it represents current HTTP request*) and queryset(*which contains the objects that user has selected*).

def clear\_inventory(self,request,queryset):

We will decorate this method with admin.action and set the description to Text we want to show in our action list

    @admin.action(description='Clear Inventory')

    def clear\_inventory(self,request,queryset):

Inside our method we can do anything with the queryset, so we can update our queryset to inventory=0

This will immediately update the database and then return the number of updated records so we can get that value and store in a variable like *updated\_count*.

    @admin.action(description='Clear Inventory')

    def clear\_inventory(self,request,queryset):

       updated\_count= queryset.update(inventory=0)

To show a message to the user, we can use self.message\_user.

    @admin.action(description='Clear Inventory')

    def clear\_inventory(self,request,queryset):

       updated\_count= queryset.update(inventory=0)

       self.message\_user()

Every model admin class contains this method for showing a message to the user. We give it two argument, First argument is request object and second argument is the actual message we want to display.

    @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 successfully updated"

        )

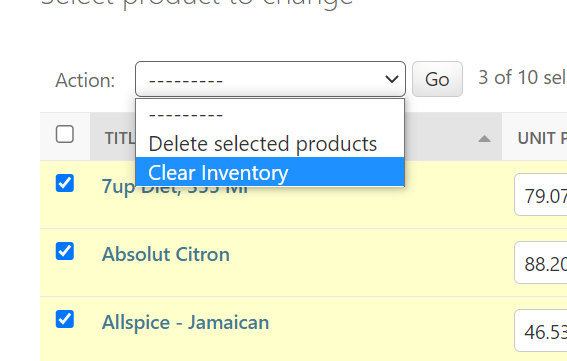
Since we have completely defined our custom action, it is time to call this action on our Product admin page.

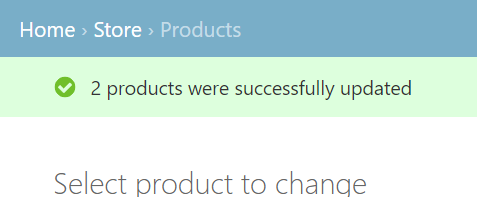
@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    actions=['clear\_inventory']

We use actions attribute of model admin class and set it to name of our custom action,

🡨We see custom action now, use it to clear inventories of some products.

🡨from self.message\_user.

Note: We can also show error message as well while calling self.message\_user() method. For this we need messages module from django.contrib

from django.contrib import admin, messages

Add a third argument in message\_user method,

    @admin.action(description="Clear Inventory")

    def clear\_inventory(self, request, queryset):

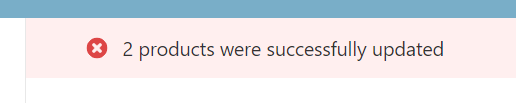
        updated\_count = queryset.update(inventory=0)

        self.message\_user(

            request, f"{updated\_count} products were successfully updated",

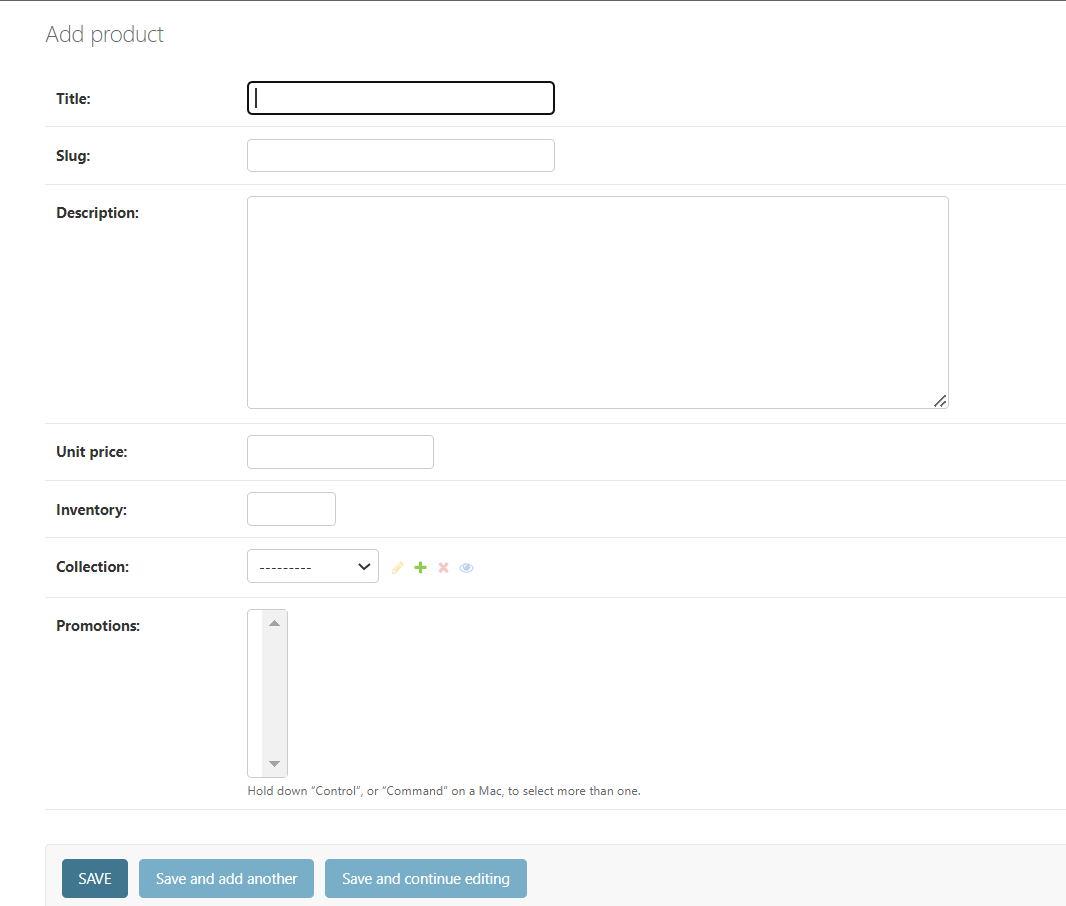
            messages.ERROR

        )



**Customizing Forms**:

We have seen a lot of customization on the list page, now let us see how can we customize the form for adding or updating models.



When we add a product, we see a beautiful form that is automatically generated based on definition of the product model.

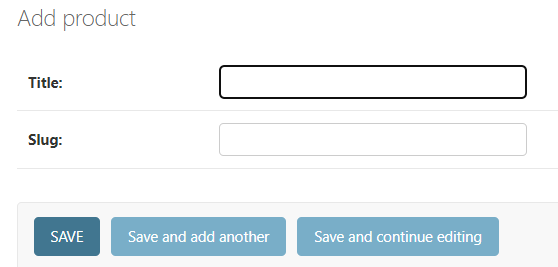
*Setting selective fields to show inside form*:

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    fields=['title','slug']

We use fields attribute.

🡨We see only 2 fields in form now.

*Excluding selective fields from display inside form*:

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    exclude = ["title", "slug"]

We use exclude attribute.

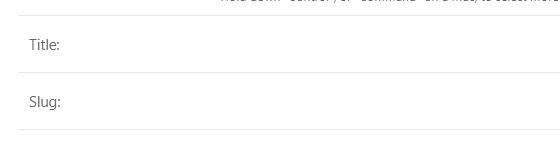
*To make certain fields read only*:

@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

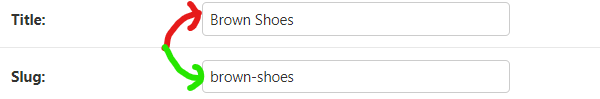
    readonly\_fields = ["title", "slug"]

We use readonly\_fields attribute



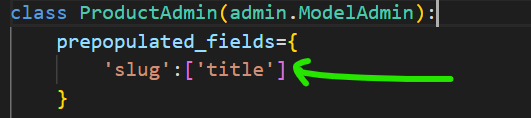
*Auto-populating fields*:

It would be nice if we type *title* and our *slug* field gets auto populated according to title.



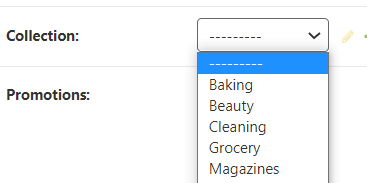
We can use prepopulated\_fields attribute and set it to a dictionary.

In this dictionary, we can specify how each field in this form can get pre-populated.

🡨Here slug field is pre-populated with title field. Since, it is set to a list, that means we can combine multiple fields here. (*Django uses some JavaScript to do this functionality*)

*Auto-complete fields*:

Here we have a collection field with a dropdown displaying all the present collections.

 Imagine, If we had 100’s or thousands of collection showing a dropdown list will have couple of user experience and performance issues.

Let us see how can we replace this dropdown with an auto complete field.

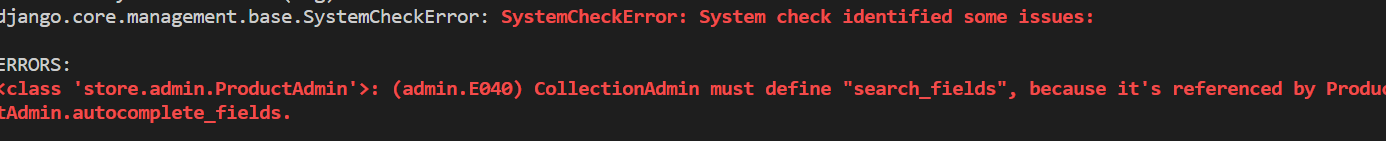
@admin.register(models.Product)

class ProductAdmin(admin.ModelAdmin):

    autocomplete\_fields=['collection']

We use autocomplete\_fields attribute and set it to list of collection

But we see an error on our terminal,



ERRORS:

<class 'store.admin.ProductAdmin'>: (admin.E040) CollectionAdmin must define "search\_fields", because it's referenced by ProductAdmin.autocomplete\_fields.

The reason for this error is that Django does not know, how to search for collections. So we need to set search\_fields attribute inside *CollectionAdmin* class.

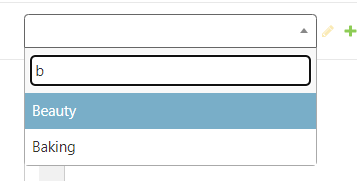
@admin.register(models.Collection)

class CollectionAdmin(admin.ModelAdmin):

    list\_display = ["title", "products\_count"]

    search\_fields = ["title"]

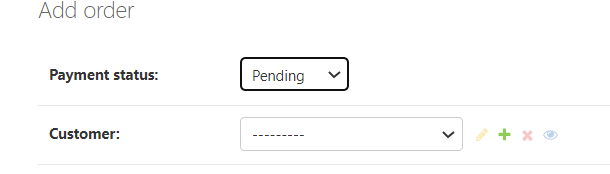
Here we set the search\_fields to title because we want to search collections by their title.



Every time we type a character, Django is going to send a request to the server to get collections whose titles match our search.

Note: If you want to find all the options available for customizing the list and forms on admin page search for django model admin and look at *ModelAdmin* options.

**Exercise**:

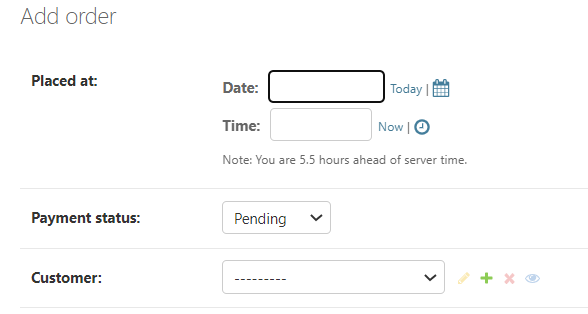


Why we see only these two fields in add order form and not placed\_at.

Solution:

placed\_at = models.DateTimeField(auto\_now\_add=True)

It is because placed\_at field is marked as being auto populated by Django. If you remove this auto\_now\_add argument, you will see

🡨placed\_at field as well

**Exercise**: Convert the dropdown customer field to autocomplete field

@admin.register(models.Order)

class OrderAdmin(admin.ModelAdmin):

    autocomplete\_fields = ["customer"]

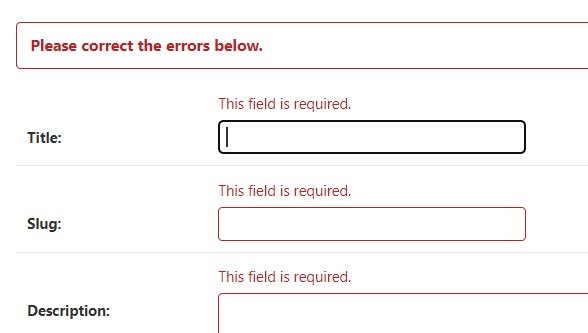
    list\_display = ["id", "placed\_at", "payment\_status", "customer"]

    list\_editable = ["payment\_status"]

**Data Validation**:

By default our forms come with basic data validation logic. Take product page for example, where we have defined given fields as required(*since none of them are nullable*).

So if we try to save the form without supplying any value, we get all these validation errors.



Let us make the Description field nullable, so back to our Product model class

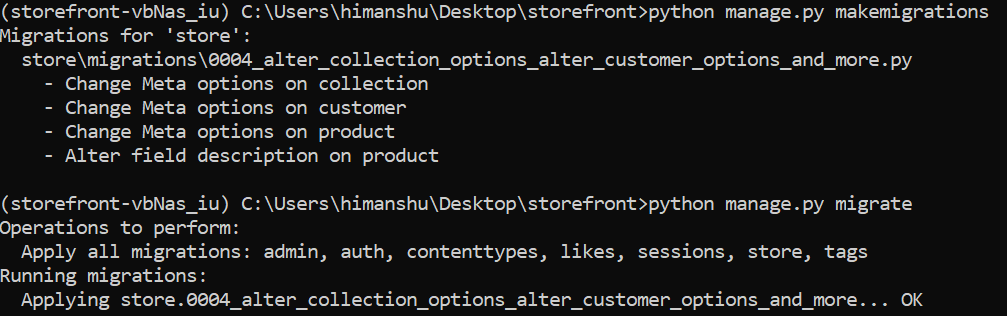
class Product(models.Model):

    title = models.CharField(max\_length=255)

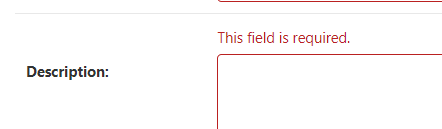
    slug = models.SlugField()

    description = models.TextField(null=True)

Run and apply migrations



Now go to the add product page and refresh it

🡨We are still getting the required field error even though we made this field nullable.

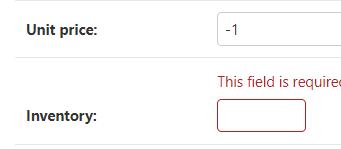
It is because null=True *argument only applies on the database*, to make this field optional in the admin interface, we should set another argument called blank=True.

description = models.TextField(null=True, blank=True)

🡨No validation error

*Using validator methods*:

Take unit price field for an example, this field type is decimal and it only accepts numbers. But what we set the unit price to negative one.

🡨validation error goes away, but this is not a valid value for unit price. So to solve this issue we will import MinValueValidator.

from django.core.validators import MinValueValidator

Note: In validators module, we have a bunch of validators. Google, Django Validators. You should see RegexValidator(For validating regular expressions), EmailValidator, URLValidator and so on.

Inside unit price field, we set validators to a list of validator objects

    unit\_price = models.DecimalField(

        max\_digits=6,

        decimal\_places=2,

        validators=[MinValueValidator(1)]

    )

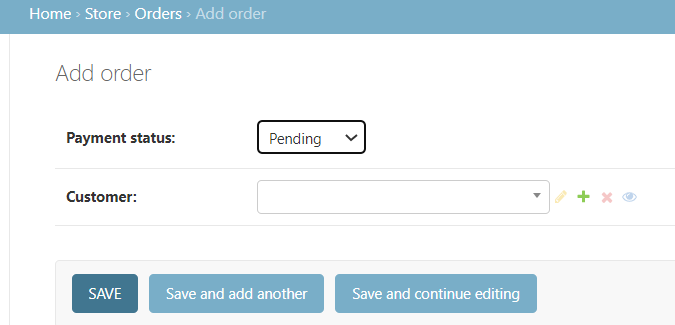
Here we *create MinValueValidator object and in the constructor we specify a minimum value as its first argument*. Optionally as a second argument we can also specify a message.

After applying changes, We get a default error message here,



**Editing Children using Inlines**:

Our current Order page



Here, we can create an order, but there is no way to edit the Order items for that particular order on the same page.

Before OrderAdmin class, define a new class called *OrderItemInline* which should inherit from either admin.TabularInline or admin.StackedInline.

class OrderItemInline(admin.TabularInline):

Inside this class we set the model to models.OrderItem.

class OrderItemInline(admin.TabularInline):

    model = models.OrderItem

We will introduce this class as an inlines to our OrderAdmin class.

@admin.register(models.Order)

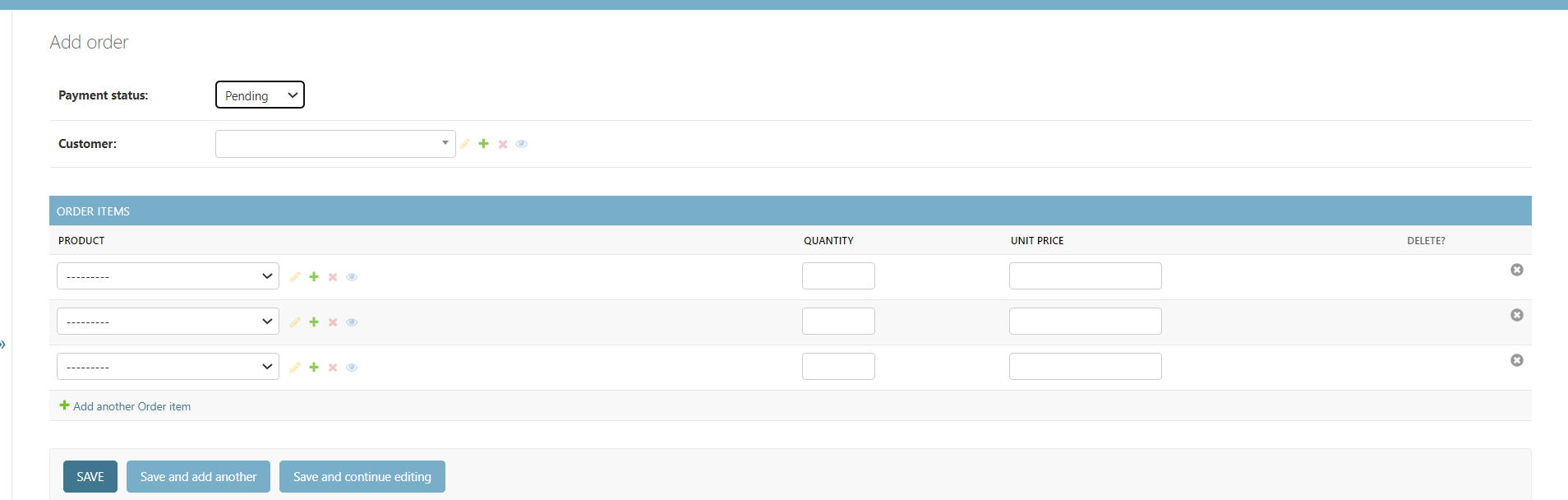
class OrderAdmin(admin.ModelAdmin):

    autocomplete\_fields = ["customer"]

    inlines = [OrderItemInline]

    list\_display = ["id", "placed\_at", "payment\_status", "customer"]

    list\_editable = ["payment\_status"]



Below the order we have this beautiful table where we manage the items for this order.

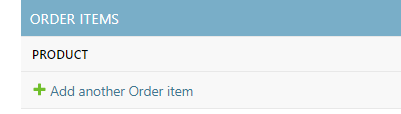
Note: If you do not want to see extra placeholders for order items then simply add extra=0

class OrderItemInline(admin.TabularInline):

    autocomplete\_fields = ["product"]

    extra = 0

    model = models.OrderItem

🡨Now by default there are no rows.

--------------------------------------------------------------------------------

We can also set minimum and maximum items for an order by setting min\_num and max\_num attributes.

class OrderItemInline(admin.TabularInline):

    autocomplete\_fields = ["product"]

    extra = 0

    max\_num=10

    min\_num=1

    model = models.OrderItem

*In case of StackedInline*:

class OrderItemInline(admin.StackedInline):

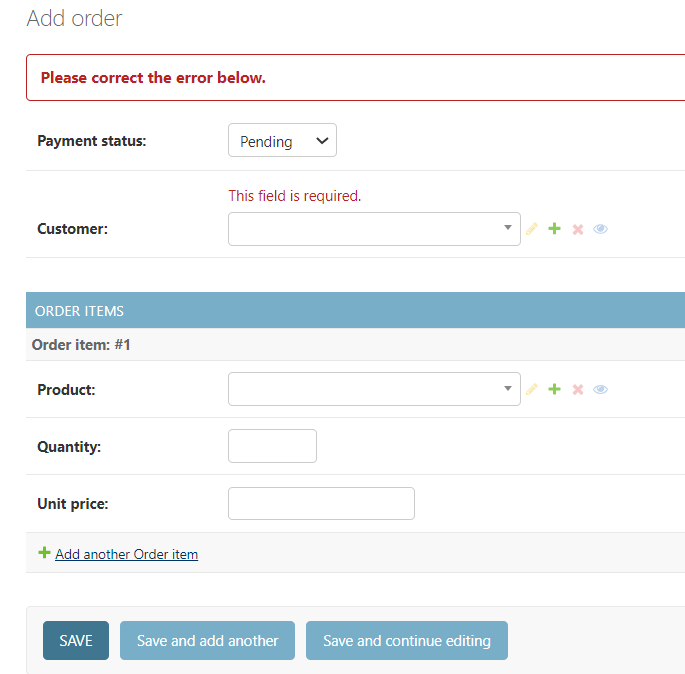
    autocomplete\_fields = ["product"]

    extra = 0

    max\_num = 10

    min\_num = 1

    model = models.OrderItem

🡨Each item will be represented as a separate form.

So this is how we can edit children using Inlines.

**Using Generic Relations**:

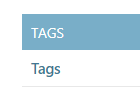
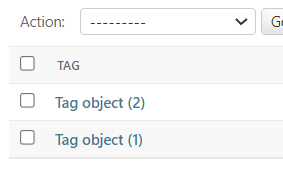
In the product form, we want to add a new section for managing the tags.

Go to tags app and in its admin.py module to register our Tag model, so that we can manage our tags in the admin interface.

from django.contrib import admin

from . import models

admin.site.register(models.Tag)

 🡪🡨We see our tags app but we need to see the tag labels. So we override it using \_\_str\_\_ method.

class Tag(models.Model):

    label = models.CharField(max\_length=255)

    def \_\_str\_\_(self) -> str:

        return self.label

Now using Inlines class we can manage tags on the Product form. So before the ProductAdmin class, create a new one called TagInline.

class TagInline():

This class will inherit from a special class called GenericTabularInline. So first import it.

from django.contrib.contenttypes.admin import GenericTabularInline

It is similar to TabularInline used earlier but for generic objects.

In this class we set model to *TaggedItem*,

class TagInline(GenericTabularInline):

    model = TaggedItem

To use this model in Product form, use inlines attribute in ProductAdmin.

@admin.register(models.Product)

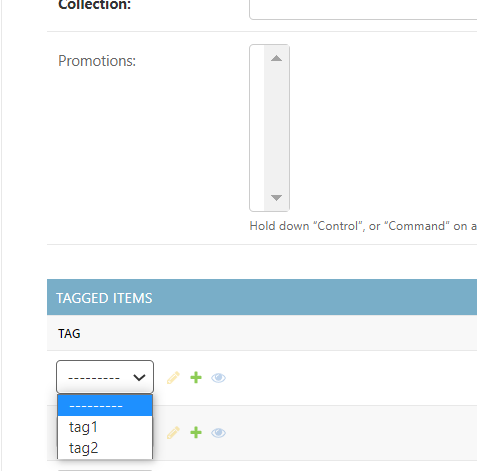
class ProductAdmin(admin.ModelAdmin):

    autocomplete\_fields = ["collection"]

    actions = ["clear\_inventory"]

    inlines = [TagInline]

Now we see tagged items in product form

🡨But we see dropdown instead of search field here. So to change that we use autocomplete fields

class TagInline(GenericTabularInline):

    autocomplete\_fields = ["tag"]

    model = TaggedItem

But we get the same error as before,

ERRORS:

<class 'store.admin.TagInline'>: (admin.E040) ModelAdmin must define "search\_fields", because it's referenced by TagInline.autocomplete\_fields.

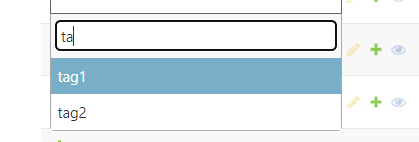
Since Django does not know how to search for tags, so we provide search\_fields in the admin module of tags app.

@admin.register(models.Tag)

class TagAdmin(admin.ModelAdmin):

    search\_fields = ["label"]

Now we have a auto complete field for searching our tags.



**Extending pluggable apps**:

In the admin module of the store app, we are importing TaggedItem class from *tags* app.

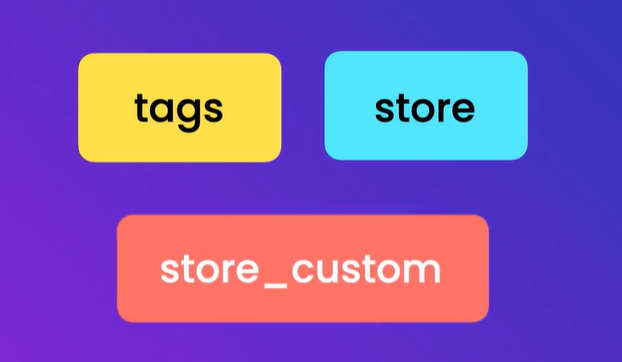
from tags.models import TaggedItem

This means our *store* app has a dependency towards *tags* app.

So we cannot build and deploy our *store* app independently.

*Ideally our apps should be self-contained, so that we can easily plug them into new projects*. So we need to ***decouple*** the store app from tags app.

So to fill gap between these two apps , we are going to create a new app called *store\_custom*.

🡨This will be the customization of the store app which know about both *tags* and *store* app.

So run python manage.py startapp store\_custom to create store\_custom app.

Navigate to admin module of the store\_custom app where we will combine features from the two pluggable apps.

from django.contrib import admin

from store.admin import ProductAdmin

from tags.models import TaggedItem

The TagInline class created in admin module of store app should also be moved here.

class TagInline(GenericTabularInline):

    autocomplete\_fields = ["tag"]

    model = TaggedItem

Now we need to create a custom ProductAdmin which extends the generic ProductAdmin that comes with our reusable app.

class CustomProductAdmin(ProductAdmin):

In this new implementation, we are going to reference TagInline class.

class CustomProductAdmin(ProductAdmin):

    inlines=[TagInline]

Since we have a new class CustomProductAdmin, we have to unregister the old one and register the new one.

*To unregister old one*:

admin.site.unregister(Product)

*Register new one with CustomProductAdmin class*:

admin.site.register(Product, CustomProductAdmin)

Finally we need to add this new app *store\_custom* into list of installed apps.

*Complete code of admin module of store\_custom app*:

from django.contrib import admin

from django.contrib.contenttypes.admin import GenericTabularInline

from store.admin import ProductAdmin

from store.models import Product

from tags.models import TaggedItem

# Register your models here.

class TagInline(GenericTabularInline):

    autocomplete\_fields = ["tag"]

    model = TaggedItem

class CustomProductAdmin(ProductAdmin):

    inlines = [TagInline]

admin.site.unregister(Product)

admin.site.register(Product, CustomProductAdmin)

We have completely replaced admin interface for Product form in store app with our store\_custom app.

If you remove store\_custom from installed app, we will see generic implementation of Product form without the tags.