Django

Backend of e-commerce

Commands:

To create django project

- pip3 install pipenv
- pipenv install django
- pipenv shell
- pipenv –venv
- django startproject store .

To create django app(django can have many apps)

- python manage.py startapp play

Writing views

- In play/view.py

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.

def say_hello(request):
    return HttpResponse('Hello')
```

Mapping view to play/urls.py

- Create urls.py in play

Pass to project urls

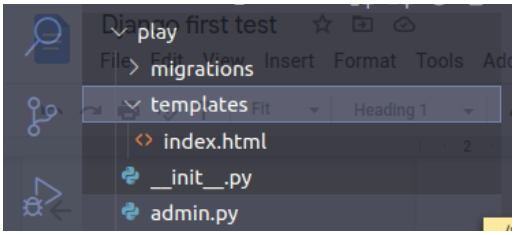
- store/urls.py

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path(('play/'), include('play.urls'))
]
```

Using template:

- First create templates folder then create .html files



- In views.py render the ‘template’ file

```
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.


def say_hello(request):
    # return HttpResponse('<h1> Hello </h1>')
    # we render
    return render(request, 'index.html')
```

- play/url

```
from django.urls import path
from . import views

urlpatterns = [
    path('index/', views.say_hello)
]
```

- store/url

```
from django.contrib import admin
from django.urls import include, path

urlpatterns = [
    path('admin/', admin.site.urls),
    path(('play/'), include('play.urls'))
]
```

- To pass by reference use dictionary in views.py

```
def say_hello(request):
    # return HttpResponse('<h1> Hello </h1>')
    # we render
    return render(request, 'index.html', {'name':
'Back'})
```

- Then in '.html' file

```
<h1>Welcome {{ name }}</h1>
```

- Also can write logic in html file

```
<!-- to write logic -->
{% if name %}
<h1>Welcome {{ name }}</h1>

{% else %}

<h1> Welcome sir</h1>

{% endif %}
```

- Debugging in vscode ????
- Django debug bar
    - steps
        1. $ pipenv install django-debug-toolbar
        2. Add to settings.py installed app

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',

'django.contrib.contenttypes',
    'django.contrib.messages',

'django.contrib.staticfiles',
    'play',
    'debug_toolbar'
]
```

3. Add url pattern in main url.py

```
urlpatterns = [
   path('admin/', admin.site.urls),
   path(('play/'),
include('play.urls')),
   path('__debug__/',
include('debug_toolbar.urls')),
]
```

4. Add the Middleware in settings.py

```
MIDDLEWARE = [

'debug_toolbar.middleware.DebugToolba
rMiddleware',
#…
]
```

5. For localhost

```
INTERNAL_IPS = [
    # ...
    "127.0.0.1",
    # ...
]
```

In our template '.html ' file to see debug tool bar we must pass proper html

Creating e-commerce model

- Apps should be as minimal as possible.
- To minimise our complexity of a project

So our models are

- Store_list
  - Product
  - Collection
  - Customer
  - Cart
  - CartItem
  - Order
  - OrderItem
- Tag
  - Tag
  - TaggedItem

Then create a model class for these apps

- In store_list app(folder) / models.py

  Model field types

  `CharField` has two extra arguments:

  CharField.max_length

  CharField.db_collation

  - We create model classes

```
class Product(models.Model):
    # model field types
    # id created automatically created by django
    title  = models.CharField(max_length=255)
    description = models.TextField()
    # let say max price is 9999.99
    price = models.DecimalField(max_digits=6,
decimal_places=2)
    inventory = models.IntegerField()
    last_update =
models.DateTimeField(auto_now=True)


class Customer(models.Model):
    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    email = models.EmailField(unique=True)
    phone = models.CharField(max_length=255)
    birth_date = models.DateField(null=True)
```

- Choice fields:

  A sequence consisting of iterables of exactly two items (e.g. **`[(A, B), (A, B) ...]`**) to use as choices for this field. If choices are given, they're enforced by model validation and the default form widget will be a select box with these choices instead of the standard text field.

  We use choice in 2 classes in customer and order.

```
class Customer(models.Model):
    MEMBERSHIP_BRONZE = 'B'
    MEMBERSHIP_SILVER = 'S'
    MEMBERSHIP_GOLD = 'G'
```

```
    MEMBERSHIP_CHOICES = [
        (MEMBERSHIP_BRONZE, 'Bronze'),
        (MEMBERSHIP_SILVER, 'Silver'),
        (MEMBERSHIP_GOLD, 'Gold')
    ]

    first_name = models.CharField(max_length=255)
    last_name = models.CharField(max_length=255)
    email = models.EmailField(unique=True)
    phone = models.CharField(max_length=255)
    birth_date = models.DateField(null=True)
    membership = models.CharField(max_length=1,
choices=MEMBERSHIP_CHOICES, default=MEMBERSHIP_BRONZE)

class Order(models.Models):
    PAYMENT_STATUS_PENDING = 'P'
    PAYMENT_STATUS_COMPLETE = 'C'
    PAYMENT_STATUS_FAILED = 'F'

    PAYMENT_STATUS_CHOICES =  [
        (PAYMENT_STATUS_PENDING, 'pending'),
        (PAYMENT_STATUS_COMPLETE, 'complete'),
        (PAYMENT_STATUS_FAILED, 'failed')
    ]


    placed_at = models.DateTimeField(auto_now_add=True)
    payment_status =models.CharField(max_length=1,
choices=PAYMENT_STATUS_CHOICES,
default=PAYMENT_STATUS_PENDING)
```

- Defining 1 to 1 relationships

  With customer and address

```
class Address(models.Model):
    street = models.CharField(max_length=255)
    city = models.CharField(max_length=255)
    customer = models.OneToOneRel(Customer,
on_delete=models.CASCADE, primary_key=True)
    # because we don't want to create id for address
that cause many to many relation
```

- Defining 1 to many relationships

```
class Collection(models.Model):
    title = models.CharField(max_length=255)

    # product = models.ForeignKey(Product, on_delete=CASCADE)
    # this should be defined in product class
class Product(models.Model):
    # model field types
    # id created automatically created by django
    title  = models.CharField(max_length=255)
    description = models.TextField()
    # let say max price is 9999.99
    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)
    # if collection deleted but not product
```

1 Collection to * Product

```
class Customer(models.Model):
      #...

      #...

class Order(models.Models):
      #...

      #...
    customer = models.ForeignKey(Customer,
on_delete=models.PROTECT)
```

1 Customer to * orders

- Many to Many

promotion to product

1. create the promotion class

```
class Promotion(models.Model):
   description =
models.CharField(max_length=255)
   discount = models.FloatField()
```

2. go to product class

```
class Product(models.Model):
   # …
   promotions =
models.ManyToManyField('Promotion')
```

- Generic relation-ship
  - Tag the tag it self
  - TagItem the tag applied to a particular item

```
class Tag(models.Model):
   label = models.CharField(max_length=255)
```

  - In Tagged Items we use generic relations instead of importing product class, because we may use tags for other needs…

```
class TaggedItem(models.Model):
   # what tag applied to what object
   # one way is that
   # from store_list.models import Product
   # product = models.ForeignKey(Product,
on_delete=models.CASCADE)
```

  - So
  - ContentType model is design for generic relation

```
from django.contrib.contenttypes.models import

class TaggedItem(models.Model):
   # type (product, video, article)
   # id
```

```
    content_type = models.ForeignKey(ContentType,
on_delete=models.CASCADE)
    object_id = models.PositiveSmallIntegerField()
```

- Another attribute is what kind of object to read the actual object that tag is applied to.

```
from django.contrib.contenttypes.fields import
GenericForeignKey

content_object = GenericForeignKey()
```

Setting up the database

- First we need migration

  $ python manage.py makemigrations

  Then it list down all migrations

  Migrations for 'store_list':
  store_list/migrations/0001_initial.py
  - Create model Cart
  - Create model Collection
  - Create model Customer
  - Create model Order
  - Create model Promotion
  - Create model Product
  - Create model OrderItem
  - Add field featured_products to collection
  - Create model CartItem
  - Create model Address

  Migrations for 'likes':
  likes/migrations/0001_initial.py
  - Create model LikedItem

  Migrations for 'tags':
  tags/migrations/0001_initial.py
  - Create model Tag
  - Create model TaggedItem

- Second running migration

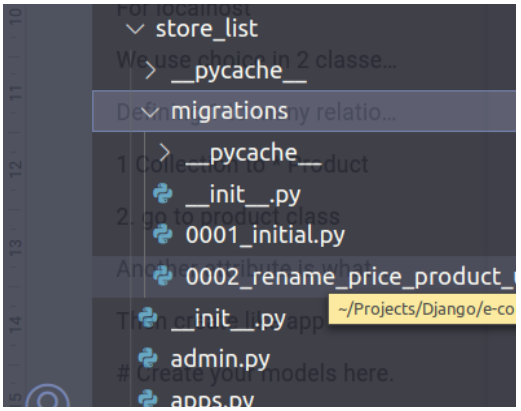
  $ python manage.py migrate

- Django creates **“primary_key=True”** in migration even if we are not assigned a primary key.
- Renaming example
  - From price to unit_price in product/model.py

```
unit_price = models.DecimalField(max_digits=6,
decimal_places=2)
```

```
(e-commerce_back-end-hLivC-og) biruk@biruk-HP-EliteBook-840-G3:~/Projects/D
thon manage.py makemigrations
Was product.price renamed to product.unit_price (a DecimalField)? [y/N] y
```

- Then it create migration history



- Add new attribute Slug

Slug is that comes after id

It's search optimization technique

Space between get max will be ignore

```
class Product(models.Model):
    # model field types
    # id created automatically created by django
    title  = models.CharField(max_length=255)
    slug = models.SlugField()
```

- Customising database schema
  - In Customer class add inner class meta reference.

```
class Customer(models.Model):
    # …

    class Meta:
        db_table = 'store_list_customer'
        indexes = [
            models.Index(fields=['last_name',
'first_name'])
        ]
```

Changes

1. Create index store_list__last_na_7e7344_idx on field(s) last_name, first_name of model customer
2. Rename table for customer to store_list_customer

    - Then makemigration
    - Then migrate

- Reverting migration
    - 1 way is basically cancel the code we write then migrate to db
    - 2nd way is when our change is large, write previous migration then it will unapply the latest one

```
❯ ✔   python manage.py migrate store 0003
Operations to perform:
  Target specific migration: 0003_add_slug_to_proc
tore
Running migrations:
  Rendering model states... DONE
  Unapplying store.0004_auto_20210608_1606... OK
```

      But the last change is in migration when we apply migration it will change again, delete code and the migration file or use git revert

- Install sql
- Install sqlclient
- Create connection
- pipenv install mysqlclient - to connect with django
- Then fix our setting

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'storefront',
        'HOST': 'localhost',
        'USER': 'root',
        'PASSWORD': ''
    }
}
```

- python .\manage.py makemigrations: says no change
- python .\manage.py migrate: it will move our to database

- python .\manage.py makemigrations store --empty
    - Under migration file the operation help us to write custom sql query

```
class Migration(migrations.Migration):

    dependencies = [
        ('store', '0002_alter_product_slug'),
    ]

    operations = [
        migrations.RunSQL("""
```

```
            INSERT INTO store_collection(title)
            VALUES ('collection1')
        """, """
            DELETE FROM store_collection
            WHERE title = 'collection1'
        """)
    ]
```

- To revert this operation we can simply write
  - python .\manage.py makemigrations store previous code

- Generate data for our database
  - https://www.mockaroo.com/

Managers and query sets

- Every django model has '.objects' to talk to our database
- '.object' returns a manager object like remote control
- Most of them returns a query set like all(),
- Query_set is an object that encapsulate a query_set - this happens when we iterate in query sets
- We can use query_set to build complex query
- In plays/views.py/function

```
    query_set = Product.objects.all()

    for product in query_set:
        print(product)

        # or
    list(query_set)
    # or
    query_set[0:4]
```

```
    query_set = Product.objects.all()

    query_set.filter()
      #this return a new query_set
      # or query_set.filter().filter().order_by()
for complex query_set
    query_set.count()
```

Retrieving object

- The 1st method is .all() method
- .get() in get we pass a lookup parameter like id and pk

```
product = Product.objects.get(pk=1)
# in get we pass a lookup parameter like id and pk
```

In sql

```
SELECT `store_product`.`id`,
       `store_product`.`title`,
       `store_product`.`slug`,
       `store_product`.`description`,
       `store_product`.`unit_price`,
       `store_product`.`inventory`,
       `store_product`.`last_update`,
       `store_product`.`collection_id`
FROM `store_product`
WHERE `store_product`.`id` = 1
```

- To avoid

  DoesNotExist at /plays/index/

  Product matching query does not exist.

  We can use

```
try:
        product = Product.objects.get(pk=1)
        # in get we pass a lookup parameter like id and pk
    except ObjectDoesNotExist:
        pass
```

But simply

```
product = Product.objects.filter(pk=0).first()
```

This will return none

Filtering objects

```
query_set = Product.objects.filter(unit_price>20)
```

- We can't write > or < we use gt or lt
- For more filed look up

```
query_set = Product.objects.filter(unit_price__gt =20)
query_set = Product.objects.filter(unit_price__range =(10, 30))

return render(request, 'index.html',  {'name':'', 'products':
list(query_set)})
```

- Then pass to our template

```
<ul>
    {% for p in products %}
    <li>{{ p.title }}</li>
```

```
        {% endfor %}
    </ul>
```

Example: `Entry.objects.get(headline__icontains='Lennon')`

SQL equivalent: `SELECT ... WHERE headline ILIKE '%Lennon%';`

```
query_set = Product.objects.filter(title__icontains='coffee')
```

This shows a list that contain coffee ‘i’ is for case insensitive

also

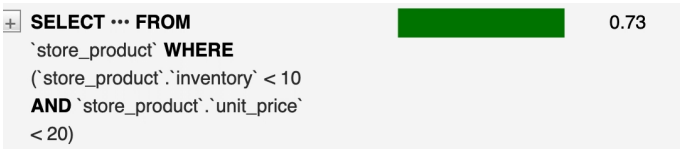
```
query_set = Product.objects.filter(last_update__year = 2021)
```

Also : **`startswith, istartswith, endswith, iendswith, date`**

Complex lookups

```
query_set = Product.objects.filter(inventory__lt=10, unit_price__lt=20)
```

Equivalent sql query is



- To use ‘or’, Q is query

```
import django.db.models import Q
# …
query_set = Product.objects.filter(Q(inventory__lt=10) |
Q(unit_price__lt=20))
```

- Referring fields with F obiect

```
query_set =Product.objects.filter(inventory=F('unit_price'))
```

- Other methods
  - .only()
  - .differ()
  - .prefetch_related()

Aggregate .Count(), .Min(), Max()

```
from django.db.models.aggregates import Count, Max, Min

query_set = Product.objects.aggregate((Count()))
```

Annotate detail

```
    Customer.objects.annotate(is_new=Value(True))
```

Add new field in db

Calling Database Functions

```
from django.db.models import Q, F, Func
  # …

def say_hello(request):

    query_set = Customer.objects.annotate(
        # CONCAT
        full_name=Func(F('first_name'), Value(' '),
                       F('last_name'), function='CONCAT')

    )
```

```
Executed SQL
  SELECT `store_customer`.`id`,
      `store_customer`.`first_name`,
      `store_customer`.`last_name`,
      `store_customer`.`email`,
      `store_customer`.`phone`,
      `store_customer`.`birth_date`,
      `store_customer`.`membership`,
      CONCAT(`store_customer`.`first_name`, ' ', `store_customer`.`last_name`) AS `full_name`
    FROM `store_customer`

Time
  2.1698474884033203 ms
```

or

```
from django.db.models.functions import Concat
#...
query_set = Customer.objects.annotate(
        # CONCAT
        full_name=Concat('first_name', Value(' '), 'last_name')

    )
```

Grouping Data

Count import from aggregate or models

from django.db.models import Count

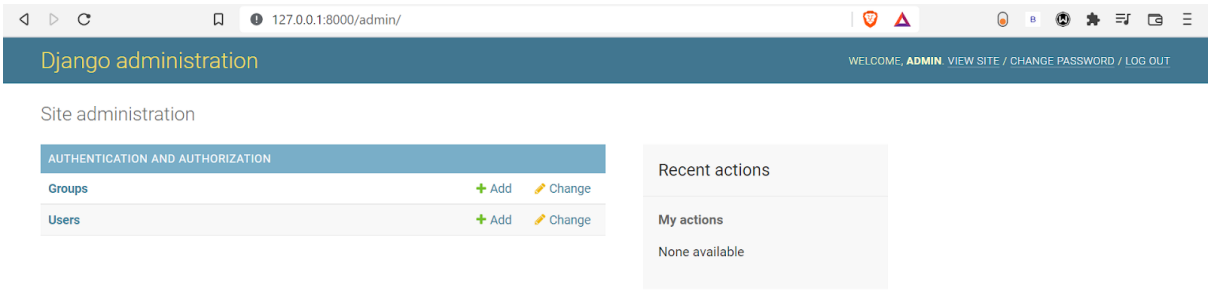
from django.db.models.aggregate import Count

```
    query_set = Customer.objects.annotate(
        orders_count=Count('order')

    )
```

```
    store_customer`.`first_name`,
    `store_customer`.`last_name`,
    `store_customer`.`email`,
    `store_customer`.`phone`,
    `store_customer`.`birth_date`,
    `store_customer`.`membership`,
    COUNT(`store_order`.`id`) AS
`orders_count`
 FROM `store_customer`
 LEFT OUTER JOIN `store_order`
   ON (`store_customer`.`id` =
`store_order`.`customer_id`)
 GROUP BY `store_customer`.`id`
 ORDER BY NULL
```

Admin site



- ☑ ~~Customising~~
- ☑ ~~Adding computed columns~~
- ☑ ~~Loading related objects~~
- ☑ ~~Adding search and filter~~
- ☑ ~~Implementing custom actions~~
- ☑ ~~Adding data validation~~

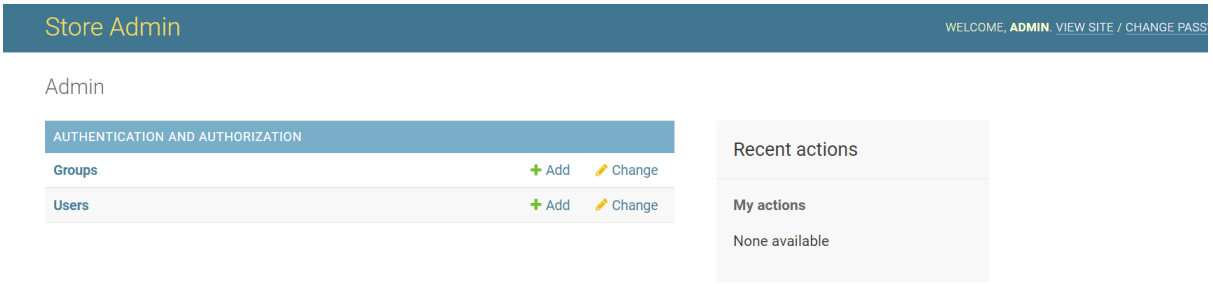
Create admin user

```
python manage.py createsuperuser
```

Generate the tables for admin information in db

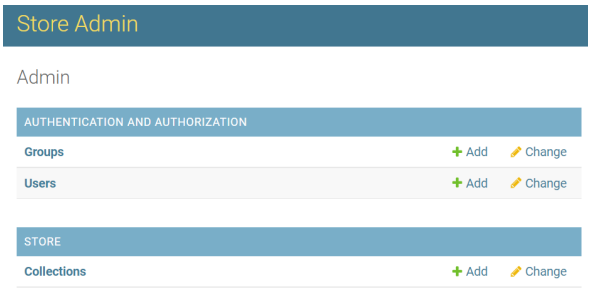
```
python manage.py migrate
```

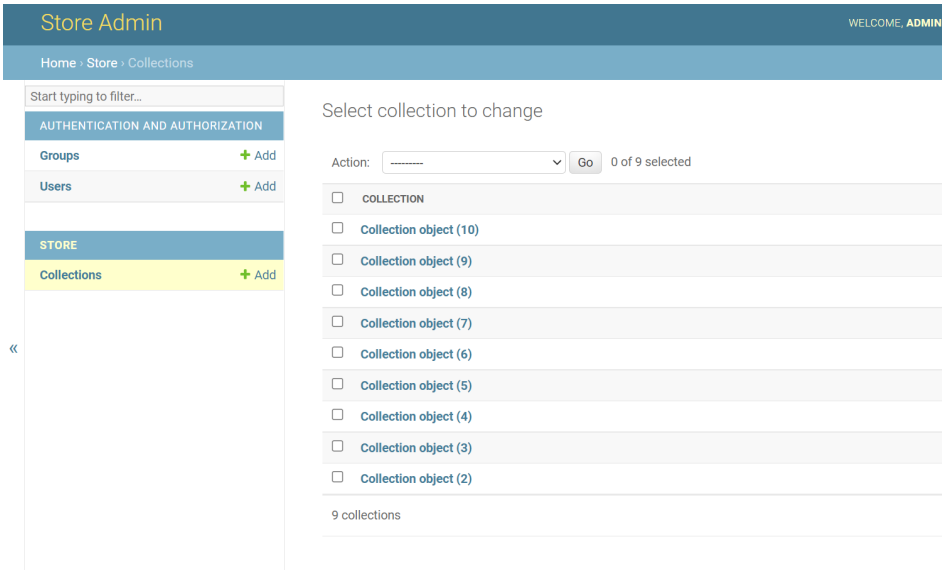
```
admin.site.site_header = 'Store Admin'
admin.site.index_title = 'Admin'
```


Store Admin
WELCOME, ADMIN. VIEW SITE / CHANGE PASS
Admin
AUTHENTICATION AND AUTHORIZATION
Groups
Add
Change
Users
Add
Change
Recent actions
My actions
None available

All customising code written under store/admin.py

- Register our models here

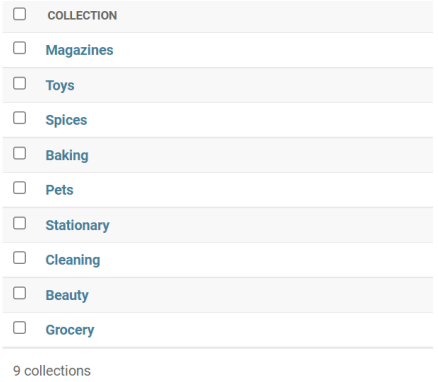




- In our store model overwrite string

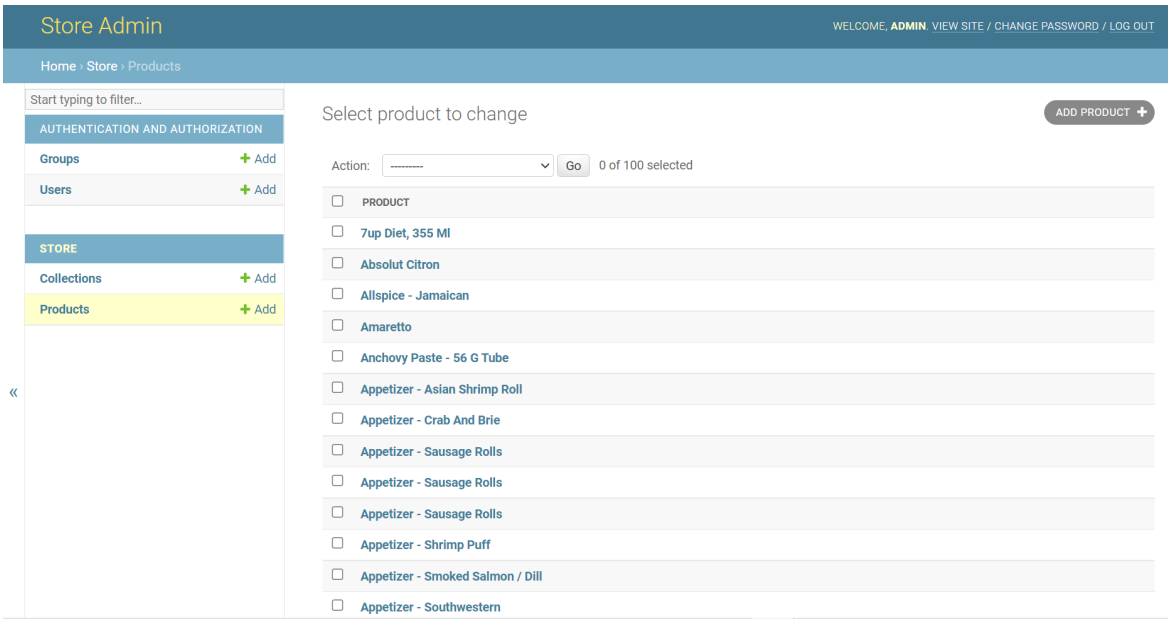
```
class Collection(models.Model):
  #...

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



- Arrange by order

```
class Collection(models.Model):
  …
    class Meta:
        ordering = ['title']
```



Customising

1 way is

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


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

Or with decorator

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


admin.site.register(models.Collection)
# admin.site.register(models.Product, ProductAdmin)
```

Adding computed columns

To add condition

```
@admin.register(models.Product)
class ProductAdmin(admin.ModelAdmin):
   …
    @admin.display(ordering='inventory')
    def inventory_status(self, product):
        if product.inventory < 10:
            return 'Low'
        return 'OK'
```

Loading related objects

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

Overriding the Base QuerySet

```
@admin.register(models.Collection)
class CollectionAdmin(admin.ModelAdmin):
   …
    @admin.display(ordering='product_count')
    def product_count(self, collection):
        return collection.product_count
    def get_queryset(self, request):
        return super().get_queryset(request).annotate(
            product_count=Count('product'))
```

Providing Links to Other Pages

```
@admin.register(models.Collection)
class CollectionAdmin(admin.ModelAdmin):
…
       return collection.product_count
```

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

    @admin.display(ordering='product_count')
    def product_count(self, collection):
       url = reverse('admin:store_product_changelist')
```

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

```
class CollectionAdmin(admin.ModelAdmin):
    list_display = ['title', 'product_count']

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

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

class CollectionAdmin(admin.ModelAdmin):
    list_display = ['title', 'product_count']

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

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

Adding Search to the List Page

```
class CustomerAdmin(admin.ModelAdmin):
    list_display = ['first_name', 'last_name', 'membership']
    list_editable = ['membership']
    ordering = ['first_name', 'last_name']
    search_fields = ['first_name', 'last_name']
```

To add start with

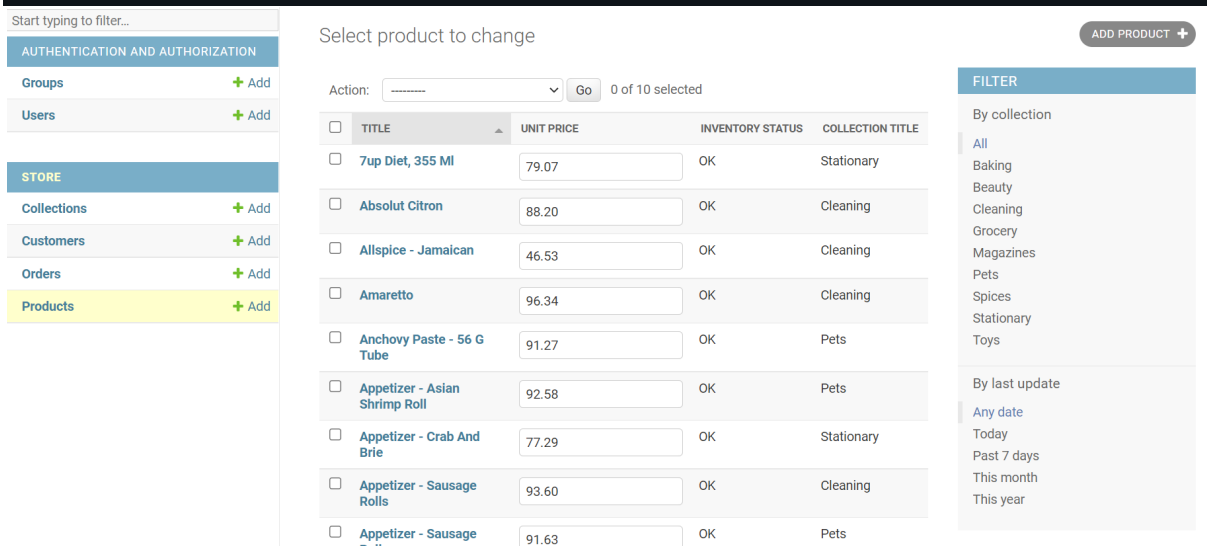
```
   search_fields = ['first_name__startswith', 'last_name__startswith']
```

To avoid case sensitive

```
    search_fields = ['first_name__istartswith',
'last_name__istartswith']
```

Filter

```
@admin.register(models.Product)
class ProductAdmin(admin.ModelAdmin):
…
    list_filter = ['collection', 'last_update']
```



Creating Custom Actions

Create class above Product admin

```
class InventoryFilter(admin.SimpleListFilter):
    title = 'inventory'
    parameter_name = 'inventor'

    def lookups(self, request, model_admin):
        return [
            ('<10', 'LOW')
        ]
    def queryset(self, request, queryset:QuerySet):
        # we write filtering logic here
        if self.value() == '<10':
            return queryset.filter(inventory__lt=10)
```

```
class InventoryFilter(admin.SimpleListFilter):
    title = 'inventory'
    parameter_name = 'inventory'

    def lookups(self, request, model_admin):
        return [
```

```
            ('<10', 'LOW')
        ]
    def queryset(self, request, queryset):
        # we write filtering logic here
        if self.value() == '<10':
            return queryset.filter(inventory__lt=10)
```

Then just add this class in list filter of ProductAdmin

(list_filter = ['collection', 'last_update', InventoryFilter])

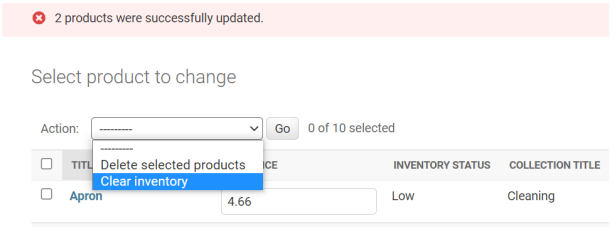
Custom action

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

…

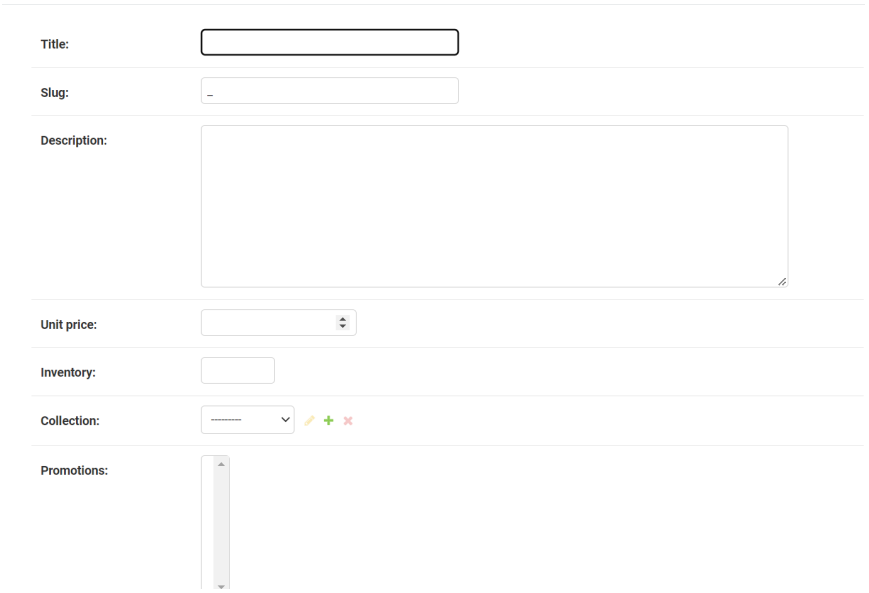
Under that class

```
    # custom action
    @admin.action(description='Clear inventory')
    def clear_inventory(self, request, queryset):
        updated_count = queryset.update(inventory=0)
        queryset.update(inventory=0)
        self.message_user(
            request,
            f'{updated_count} products were successfully updated.',
            # messages.ERROR
            messages.ERROR )
```



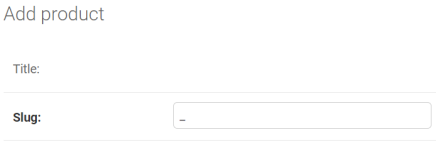
Customise form

By default



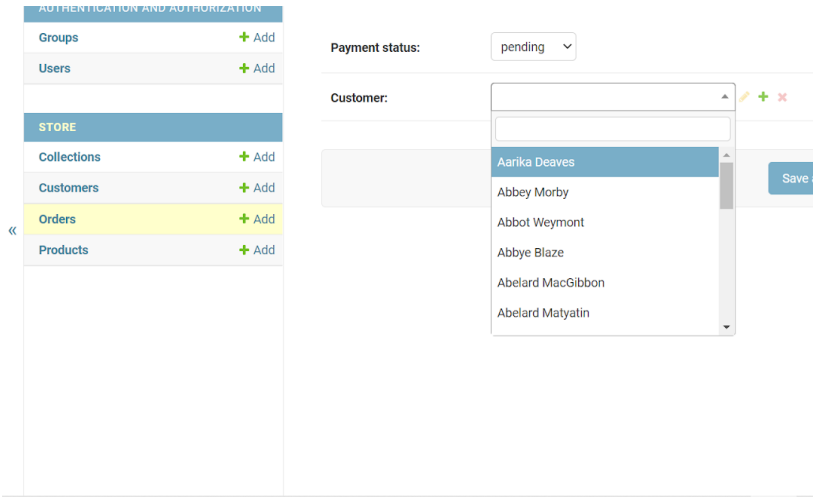
If

```
@admin.register(models.Product)
class ProductAdmin(admin.ModelAdmin):
    fields = ['title', 'slug']
    readonly_fields = ['title']
```



Also order

```
@admin.register(models.Order)
class OrderAdmin(admin.ModelAdmin):
    list_display = ['id', 'placed_at', 'customer']
    autocomplete_fields = ['customer']
```



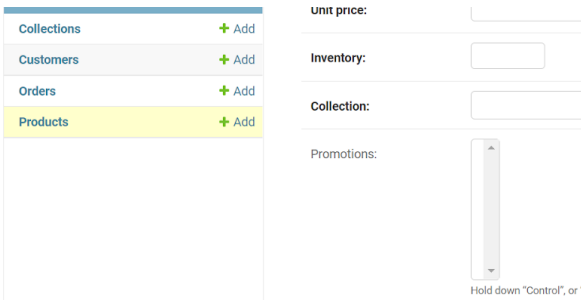
Adding Data Validation

It has by default

More on [Validators]

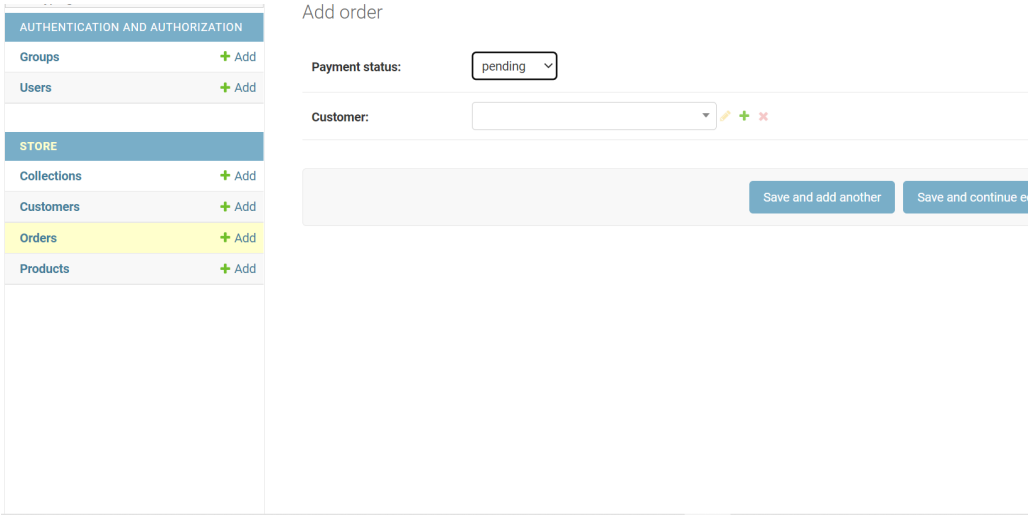
```
# let say max price is 9999.99
unit_price = models.DecimalField(
    max_digits=6,
    decimal_places=2,
    validators=[MinValueValidator(1)])
```

```
promotions = models.ManyToManyField('Promotion', blank=True)
```

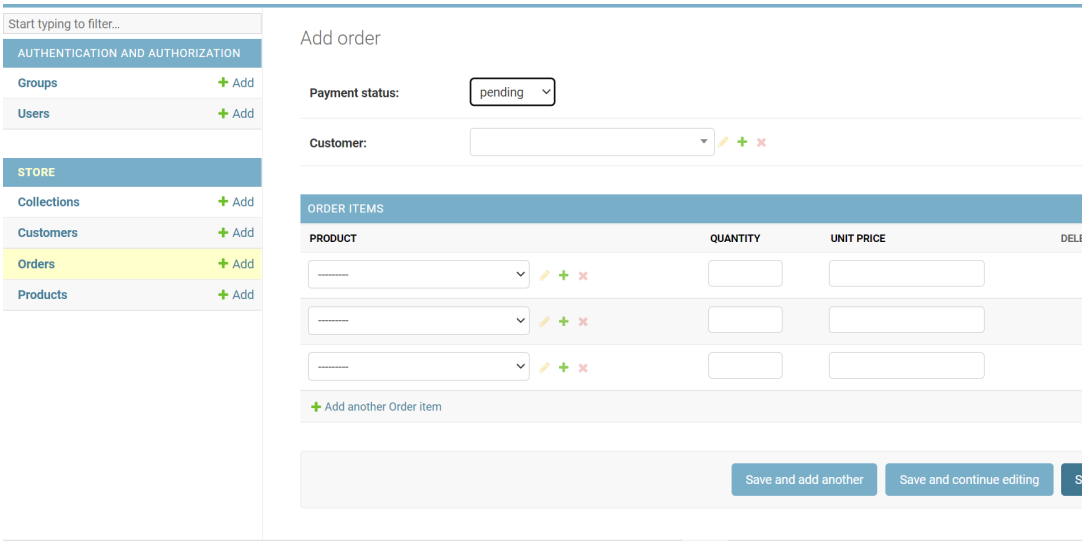


Editing children

By default



Then this



Code is

```
class OrderItemInline(admin.TabularInline):
    autocomplete_fields = ['product']
    model = models.OrderItem

@admin.register(models.Order)
class OrderAdmin(admin.ModelAdmin):

    autocomplete_fields = ['customer']
    inlines =[OrderItemInline]
    list_display = ['id', 'placed_at', 'customer']
```

Also create store_custom app

REST-FRAMEWORK

[Documentation]

What are restful api
Representational state transfer(Rest): is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web.

Our endpoint apis may be

- /orders,
- /products,
- /carts

These endpoints sent or response when we get a request from client side

Restfulapi or rest has rules

Benefits :

- Fast
- Scalable
- Reliable
- Easy to understand
- Easy to change

There are 3 basic rules

1. Resource
2. Representation
3. Http methods

1. Resource

Resources are the basic building block of a RESTful service. Examples of a resource from an online book store application include a book, an order from a store, and a collection of users.

Resources are addressable by URLs and HTTP methods can perform operations on resources. Resources can have multiple representations that use different formats such as XML and JSON. we can use HTTP headers and parameters to pass extra information that is relevant to the request and response.

In our case

- /orders,
- /products,
- /carts

How to allocate them

1. Representation

Representation of Resources

A resource in REST is a similar Object in Object Oriented Programming or is like an Entity in a Database.

Once a resource is identified then its representation is to be decided using a standard format

Represent as

- Html
- Xml
- Json(JavaScript Object Notation) : is a lightweight data-interchange format. It is easy for humans to read and write.

2. Http methods

What does a client want to do ?

The primary or most-commonly-used HTTP verbs (or methods, as they are properly called) are POST, GET, PUT, PATCH, and DELETE.

These correspond to create, read, update, and delete (or CRUD) operations, respectively.

HTTP works as a request-response protocol between a client and server.

- GET: gets resource
- POST: create
- PUT: all property update
- PATCH: specific property update
- DELETE: specific property update

Django rest-framework installation

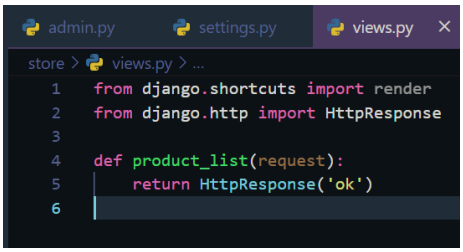
These is a separate framework than django sits up on django framework and make easy to build an api

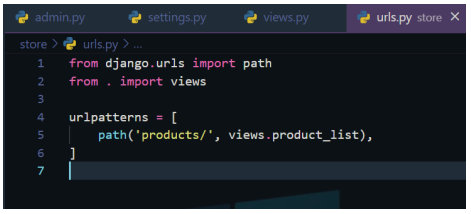
1. pipenv install djangorestframework
2. Add it to settings.py

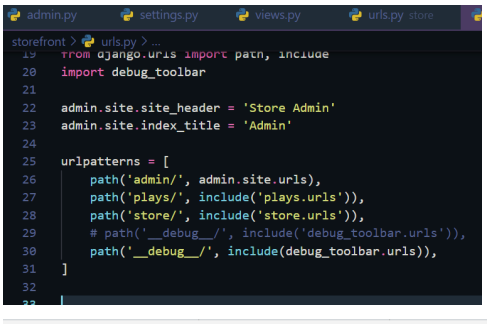
```
INSTALLED_APPS = [
    ...
    'django.contrib.staticfiles',
    'rest_framework',
```

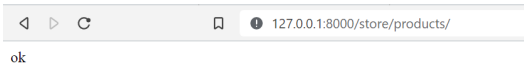
Creating api views

In django we create









Django	Rest framework
Httprequest	Request
Httpresponse	response