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

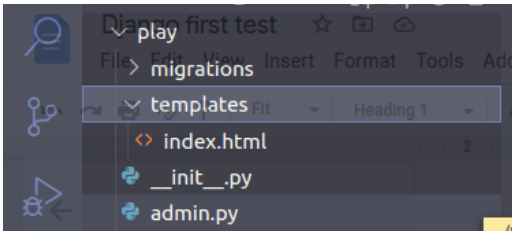
- store/urls.py

```
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

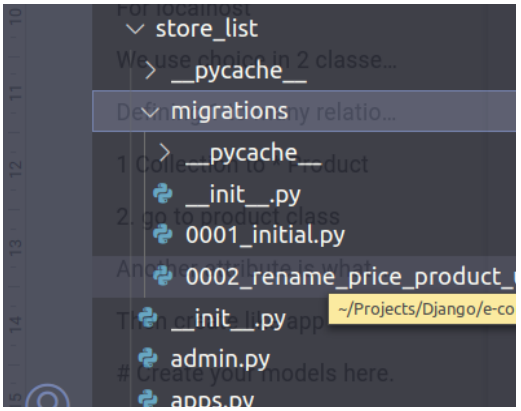
  $ python manage.py migrate

- 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]
```

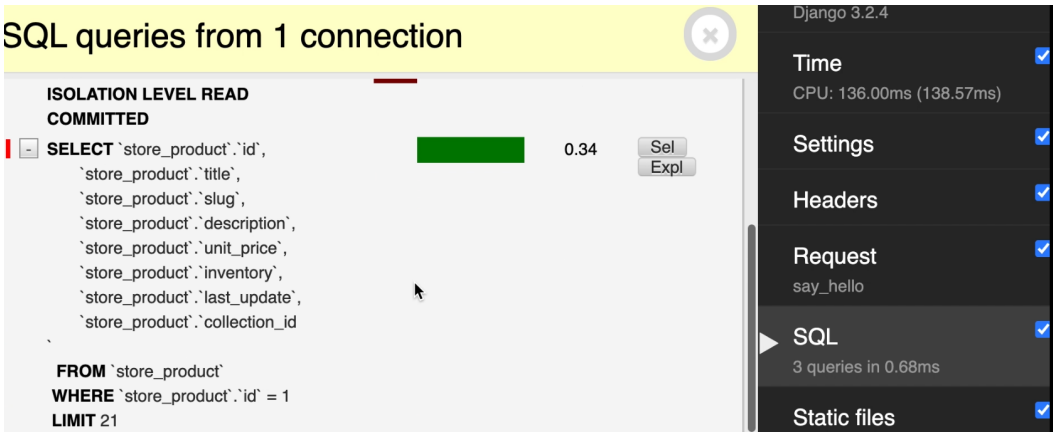
```
    query_set = Product.objects.all()

    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
```



- 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