If We Re-Name The Fields Inside Any Model:

```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>python manage.py makemigrations
Was product.price renamed to product.unit_price (a DecimalField)? [y/N] y
Migrations for 'store':
  store\migrations\0002_rename_price_product_unit_price.py
    ~ Rename field price on product to unit_price
```

****

After We Add New Field To Model Called Slug:

```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>python manage.py makemigrations
It is impossible to add a non-nullable field 'slug' to product without specifying a default. This is because the database needs something to populate existing rows.
Please select a fix:
 1) Provide a one-off default now (will be set on all existing rows with a null value for this column)
 2) Quit and manually define a default value in models.py.
Select an option: 1
Please enter the default value as valid Python.
The datetime and django.utils.timezone modules are available, so it is possible to provide e.g. timezone.now as a value.
Type 'exit' to exit this prompt
>>> '-'
Migrations for 'store':
  store\migrations\0003_product_slug.py
    + Add field slug to product

(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>
```

```python
class Product(models.Model):
    title = models.CharField(max_length=255, unique=True);

    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(to=Collection, on_delete=models.PROTECT);

    # promotions = models.ManyToManyField(to=Promotion, related_name='products');
    promotions = models.ManyToManyField(to=Promotion);
```

****

To Run Migrations That We Created:

- Run Command: *python manage.py migrate*

```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, likes, sessions, store, tags
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying likes.0001_initial... OK
  Applying sessions.0001_initial... OK
  Applying store.0001_initial... OK
  Applying store.0002_rename_price_product_unit_price... OK
  Applying store.0003_product_slug... OK
  Applying tags.0001_initial... OK
```

****

To Show The SQL That Generated From Specific Migrations:

- Run Command: *python manage.py sqlmigrate store 0003*
    - *Here store is the app*
    - *And, 0003 is The Number Of Migration*

```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>python manage.py sqlmigrate store 0003
BEGIN;
--
-- Add field slug to product
--
CREATE TABLE "new__store_product" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "slug" varchar(50) NOT NULL, "title" varchar(255) NOT NULL UNIQUE, "description" tex
t NOT NULL, "inventory" integer NOT NULL, "last_update" datetime NOT NULL, "collection_id" bigint NOT NULL REFERENCES "store_collection" ("id") DEFERRABLE INITIALLY DEF
ERRED, "unit_price" decimal NOT NULL);
INSERT INTO "new__store_product" ("id", "title", "description", "inventory", "last_update", "collection_id", "unit_price", "slug") SELECT "id", "title", "description",
"inventory", "last_update", "collection_id", "unit_price", '-' FROM "store_product";
DROP TABLE "store_product";
ALTER TABLE "new__store_product" RENAME TO "store_product";
CREATE INDEX "store_product_slug_6de8ee4b" ON "store_product" ("slug");
CREATE INDEX "store_product_collection_id_2914d2ba" ON "store_product" ("collection_id");
COMMIT;
```

****

To Add MetaData To Our Model:

- First Create **Meta-Class** Inside The Model Class.
- Then Define The Meta That We Want

```
class Customer(models.Model):
      ...Here We Define Our Fields...
    class Meta:
        db_table = 'store_customers'; # Here we Define The Name Of Table.
        indexes = [
            models.Index(fields=['last_name', 'first_name'])
        ]
```

```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>python manage.py makemigrations
Migrations for 'store':
  store\migrations\0004_customer_store_custo_last_na_e6a359_idx_and_more.py
    + Create index store_custo_last_na_e6a359_idx on field(s) last_name, first_name of model customer
    ~ Rename table for customer to store_customers
```

```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, likes, sessions, store, tags
Running migrations:
  Applying store.0004_customer_store_custo_last_na_e6a359_idx_and_more... OK
```

****

To Re-Applying The Migrations To Specific Point:

- Run Command: *python manage.py migrate app-name number-here*
    - Ex: *python manage.py manage.py migrate store 0003*

```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>python manage.py migrate store 0003
Operations to perform:
  Target specific migration: 0003_product_slug, from store
Running migrations:
  Rendering model states... DONE
  Unapplying store.0004_customer_store_custo_last_na_e6a359_idx_and_more... OK
```

****

To Use MySQL With Django:

- Install The mysqlclient: *pip install mysqlclient*

```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>pip install mysqlclient
Collecting mysqlclient
  Downloading mysqlclient-2.2.6-cp311-cp311-win_amd64.whl.metadata (4.8 kB)
Downloading mysqlclient-2.2.6-cp311-cp311-win_amd64.whl (207 kB)
Installing collected packages: mysqlclient
Successfully installed mysqlclient-2.2.6
```

****

The Default DB For Django Is: SQLite3:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

****

```
DATABASES = {
    'default': {
        # 'ENGINE': 'django.db.backends.sqlite3',
        # 'NAME': BASE_DIR / 'db.sqlite3',

        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'storefront',
        'HOST': 'localhost',
        'USER': 'root',
        'PASSWORD': '123'
    }
}
```

****

To Create Empty Migration To Use Custom SQL:

```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>python manage.py makemigrations store --empty
Migrations for 'store':
  store\migrations\0004_auto_20241213_0926.py
```

****

To Run Custom SQL From Migration File:

- Note 1: The First Param is → Run The SQL Command.
- Note 2: The Second Param is → To Re-do The Operation.

```
class Migration(migrations.Migration):

    dependencies = [
        ('store', '0003_product_slug'),
    ]

    operations = [
        migrations.RunSQL("""
            INSERT INTO store_collection (title)
            VALUES ('collection1');
        """, """
            DELETE FROM store_collection
            WHERE title='collection1'
        """)
    ]
```

****

To Run The Migration File:

```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, likes, sessions, store, tags
Running migrations:
  Applying store.0004_auto_20241213_0926... OK
```

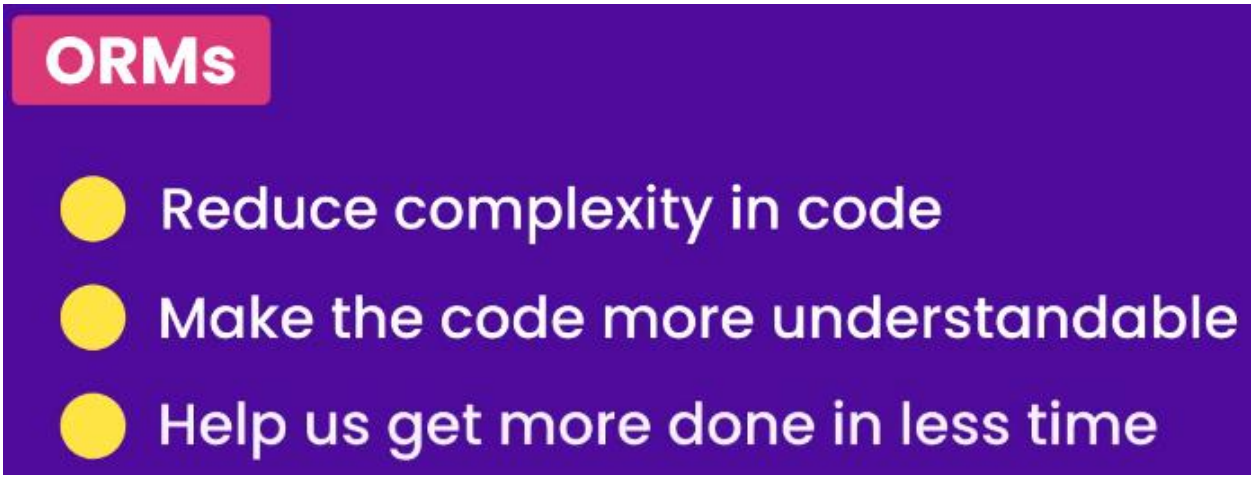
****

https://mockaroo.com/

****

```
(.venv) G:\Web\Django\Code-With-Mosh-Ultimate-Django-Series>mysql -u root -p storefront < store_customer.sql
```

****



****

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

    for product in query_set:
        print("The Product Is: ", product);
```

****

To Get Data By Id:

Note 1: The Benefit Of Using pk That Are We Query The Data By Id-Column Not By Name; It May Be Code, title, ...etc.

Note 2: The get-method will return The Object Not Query Set.

Note 3: If The Object Not Found It Will Throw An Exception.

```
from django.core.exceptions import ObjectDoesNotExist;

product = Product.objects.get(id=1);

product_2 = Product.objects.get(pk=2);

print("The Query Data Is: ", product.title);
print("The Query Data With Pk is: ", product_2.title);
```

****

Note 1: In This Way, We Don't Have Exception, We Will Get None.

```
product = Product.objects.filter(pk=3).first();

    if product is None:
        print("No Product Found");
print("The Product Data Is: ", product.description);
```

****

In This Way We Can Filter Data By Using gt, lt, lte, gte:

```
product_query_set = Product.objects.filter(unit_price__gt=20);
product_query_set_02 = Product.objects.filter(unit_price__lte=20);
```

****

Note: Here We Use Tuple To Get The Values In Range:

```
product_query_set_03 = Product.objects.filter(unit_price__range=(20, 30));
```

****

```
{% if products %}

    {% for product in products %}
        <li>{{ product.title }}</li>
        <li>{{ product.description }}</li>
        <hr>
    {% endfor %}

{% endif %}
```

****

```
# product_query_set_with_collection =
Product.objects.filter(collection_id=2).query;

product_query_set_with_collection = Product.objects.filter(collection__id=6);

# product_query_set_with_collection =
Product.objects.filter(collection__id__range=(4, 6));
```

****

Note: This Search Is Case Sensitive.

```
product_query_set_with_collection =
Product.objects.filter(title__contains='fruit');
```

****

Note: In This Way The Search Is Case In-sensitive.

```
product_query_set_with_collection =
Product.objects.filter(title__icontains='coffee');
```

****

```
# To Filter Depending On The Value Of Specific Attribute Of The Field:

product_query_set_with_collection =
Product.objects.filter(last_update__year=2021);
```

****

```
product_query_set= Product.objects.filter(description__isnull=True);
```

****

To Query Products Using And-Operator:

```
products = Product.objects.filter(inventory__lt=10, unit_price__gt=20);
```

****

To Query The Products Using OR-Operator:

Note 1: Here We Use **Q** For **Query Expressions**, Where We Can Use Bitwise Operator.

Note 2: Each Q Expression Can Capsulate One Expression Only.

Note 3: Here We Can Also Use AND-Operator Using &.

```
from django.db.models import Q;
products_query_set = Product.objects.filter(Q(inventory__lt=10) |
Q(unit_price__lt=20));
```

****

In This Way We Can Negate The Operator Of Q:

```
products_query_set = Product.objects.filter(Q(inventory__lt=10) |
~Q(unit_price__lt=20));
```

****

To Reference A Field In The Model, We Can Use F-class:

```
from django.db.models import Q, F;
# products_query_set = Product.objects.filter(inventory=F('unit_price'));
products_query_set = Product.objects.filter(inventory=F('collection__id'));
**********************************************************************************
```

To Order Data:

Note 1: Here We Order By unit_price In ASC.

Note 2: If We Have 2-Objects OR More Have The Same Unit Price Then We Order By title In DESC;

```
products_query_set = Product.objects.order_by('unit_price', '-title');

**********************************************************************************
```

To Reverse The Results Of Ordering:

```
products_query_set = Product.objects.order_by('unit_price', '-title').reverse();
**********************************************************************************
```

To Chain The Queries:

```
products_query_set =
Product.objects.filter(collection__id=6).order_by('unit_price');
**********************************************************************************
```

If We Are Interested In One Object From Ordering:

Note 1: In This Way We Avoid QuerySet.

Note 2: In This Way We Sort The Data By unit_price, Then Get The First Object Not QuerySet Object.

```
product = Product.objects.earliest('unit_price');
print("The Product Title Is: ", product.title);
**********************************************************************************
```

Same Above But We Get The Last Object:

```
product_2 = Product.objects.latest('unit_price');
print("The Product-02 Title Is: ", product_2.title);
**********************************************************************************
```

To Limit the Results That We Are Get (Here We Get The First 5-Elements):

Note: Here We Get Products Using Indexes: 0, 1, 2, 3, 4 → This Means 5 Will Excluded

```
products_query_set = Product.objects.order_by('unit_price', '-title')[:5];
```
****

To Implement Skip And Limit:

```
products_query_set = Product.objects.order_by('unit_price', '-title')[5:10];
```
****

To Get Specific Fields From Specific Table, We Can Use values OR values_list Method:

```
# This Will Return Dict When Evaluated
products_query_set = Product.objects.values('id', 'title', 'collection__title')
```
****

```
# This Will Return Tuple Of Values When Evaluated
# id --> result_tuple[0]
# title --> result_tuple[1]
# collection__title --> result_tuple[2]
products_query_set = Product.objects.values_list('id', 'title',
'collection__title')
```
****

Note: Also We Can use **only(...Fields Here...)-Method** But If We Access Fields That Are Not Listed In Fields Then Will End With Thousands Of Queries That May Break Down The Application.

****

To Make Subquery OR Nested Queries For Filtering:

```
# When We Create Relation Between OrderItem And Product, Django Will Create
# product_id Column At Runtime.
    order_item_queryset = OrderItem.objects.values('product_id').distinct()

# order_item_queryset = OrderItem.objects.values('product__id').distinct()
products_queryset = Product.objects.values('id', 'title')\
        .filter(id__in=order_item_queryset)\
        .order_by('title')
```
****

If We Want To Select The Related Relation For Specific Model, We Can Use **select_related-Method**:

```
products_queryset = Product.objects.select_related('collection').all()
**********************************************************************************
```

And To Access The Related Model From Serializer OR Template:

```
    {% for product in products_list %}
        <li>{{ product.title }}-->{{ product.collection.title}}</li>

        <hr />
    {% endfor %}
**********************************************************************************
```

Note: The Problem With select_related Is That It Return Only One Related Model, If We Want To Return All Models For Relations Like Many-To-Many, We Can Use **prefetch_related-Method**:

```
products_queryset = Product.objects.prefetch_related('promotions').all()
print("The Query Using prefetch_related Is: ", products_queryset)
**********************************************************************************
```

We Can Mix **select_related-Method** With **prefetch_related-Method** To Get The Results:

Note: Here orderitem_set__product Is Another Relation That We Want To Select All Its Products.

```
# orderitem_set Will Be Created From Django AS Reverse Of Relationships

orders_queryset = Order.objects\
        .select_related('customer')\
        .prefetch_related('orderitem_set__product')\
        .order_by('-placed_at')[:5]
**********************************************************************************
```

If We Want To Get The prefetch_related-Method Result, Then We Must Use **all()-method** Inside The Serializer OR Inside The Template:

```
{% if orders %}

    {% for order in orders %}
        <li>{{ order.payment_status }}-->{{ order.customer.first_name }}</li>

        {% for item in order.orderitem_set.all %}
            <li>Product Title Is: {{ item.product.title }}</li>
        {% endfor %}

        <hr />
    {% endfor %}

{% endif %}
```

****