django

Wednesday, February 9, 2022 12:36 AM

Commands	output
section.1.Django Fundamentals 1. Introduction 2. Start Server 3. Django default apps 4. Create an app 5. Registering apps 6. view(works like request handler) 7. Mapping URLs to views 8. Django-templates **section.2.Building a Data Mode** 1. Data Model 2. Creating Models: 3. choice fields 4. One-to-one relationship 5. on_delete 6. One-to-many Relationship 7. One-to-many relationship of entire data model: 8. Many-to-Many relationship: 9. Resolving Circular Relationships(circular dependency): 10. Generic Relationships **section.3.Setting Up the Database** **section.4.Django ORM**	
Introduction storefront is our project and playground is our app. django-admin startproject <project-name> .	creates project files at the current location storefront ├ asgi.py ├ settings.py ├ urls.py ├ wsgi.py └ __init__.py
Start Server python manage.py runserver # manage.py knows our settings	start server
Django default apps 'django.contrib.admin', #this app provides the admin interface for managing our data 'django.contrib.auth',#this is used for authentication of users 'django.contrib.contenttypes',# # 'django.contrib.sessions', #this is used for managing data on server it is notused these days. 'django.contrib.messages',#displaying one time notification to users. 'django.contrib.staticfiles',# this provides static files to user like images.	
Create an app python manage.py startapp <playground> #this creates app	
Registering apps Everytime you create an app register that app in the settings/INSTALLED_APPS list. INSTALLED_APPS = ['django.contrib.admin', 'django.contrib.auth', 'django.contrib.contenttypes', # 'django.contrib.sessions', 'django.contrib.messages', 'django.contrib.staticfiles', **'playground'**]	
view(works like request handler) It takes a **request** and returns a **response**.It is also called request handler.	
Mapping URLs to views (views.py) from django.shortcuts import render from django.http import HttpResponse # Create your views here.	http://127.0.0.1:8000/playground/hello/ Hello World!

```
def say_hello(requests):
    return HttpResponse('Hello World!')

(urls.py)>app
from django.urls import path
from . import views
#URLConf
urlpatterns = [
    path('hello/', views.say_hello)
]

(urls.py)>project
"""storefront URL Configuration
The `urlpatterns` list routes URLs to views. For more information please
see:
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include,
path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path,include
urlpatterns = [
    path('admin/', admin.site.urls),
    path('playground/',include('playground.urls'))
]
```

Django-templates

View in other frameworks is called template in django.

creates new folder in ./playground/templates

(views.py)

```
from urllib import request
from django.shortcuts import render
from django.http import HttpResponse
# Create your views here.
def say_hello(request):
    return render(request, 'hello.html',{'name':'Django'})
```

(hello.html)

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

```
playground
├── migrations
│   ├── __pycache__
│   │   └── __init__.cpython-37.pyc
│   └── __init__.py
├── templates
│   └── hello.html
├── __pycache__
│   ├── admin.cpython-37.pyc
│   ├── apps.cpython-37.pyc
│   ├── models.cpython-37.pyc
│   ├── urls.cpython-37.pyc
│   ├── views.cpython-37.pyc
│   └── __init__.cpython-37.pyc
├── admin.py
├── apps.py
├── models.py
├── tests.py
├── urls.py
├── views.py
└── __init__.py
```

section.2.Building a Data Model

Data Model:
Tables joined together with foreign keys. Here tables are called entities.

Design your own data models according to your need with one-to-one,one-to-many,may-to-many relations.

Avoid Monolith app:
Don’t put all models in a single app rather make multiple apps and separate them.

Avoid coupling(minimal coupling or high cohesion/focus):
There should be minimum coupling between tables.And each app should handle one functionality properly.

Creating Models:
Django Field types:

- CharField
- TextField
- DecimalField
- IntegerField
- DateTimeField #give the exact time.
- DateField #gives the day not the exact time.
- EmailField
- PositiveSmallIntegerField

```
from django.db import models

#creating product model
class Product(models.Model):
    title=models.CharField(max_length=255)
    description=models.TextField()
    price=models.DecimalField(max_digits=6,decimal_places=2)
    inventory=models.IntegerField()
```

```
    last_update=models.DateTimeField(auto_now=True)
```

"auto_now_add":
will set time when an instance is created

"auto_now":
will set time when someone modified his feedback.

By default django automatically creates an **"id" field which is primary key for that table.** If you want to set your own primary key then define your column and make it primary key using **"primary_key=True"**

choice fields
This field type is used to set the **"possible input values"** for a field.
e.g:The membership can be gold, silver and bronze.

```
MEMBERSHIP_BRONZE='B'
MEMBERSHIP_SILVER='S'
MEMBERSHIP_GOLD='G'

MEMBERSHIP_CHOICE=[
    ("B","Bronze"),
    ("S","Silver"),
    ("G","Gold")
]

# this field only takes 3 values either 'B','S' or 'G'

membership=models.CharField(max_length=1,
choices=MEMBERSHIP_CHOICE,default=MEMBERSHIP_BRONZE)
```

One-to-one relationship

You only have to create relationship in one class django automatically create reverse relationship in other class.

Order of parent and child class sequence matter in code. Parent classes should be on top and child classes should be below parent classes.
But sometimes you cant follow this so **pass models name as string then**.e.g

```
    collection = models.ForeignKey('Collection', on_delete=models.PROTECT)

#relationship of customer with adresses.
ass Address(models.Model):
    street=models.CharField(max_length=255)
    city=models.CharField(max_length=255)
    customer=models.OneToOneField(Customer,on_delete=models.CASCADE,primar
y_key=True)
```

on_delete

```
on_delete=models.PROTECT # prevent deleting the child table fields
on_delete=models.CASCADE # delete all relating fields in other tables
on_delete=models.SET_NULL # set the relating fields to null.
on_delete=models.SET_DEFAULT
```

One-to-many Relationship

```
# customer(1)----------Adresses(*)
```
A customer can have many addresses.

```
class Customer(models.Model):
    MEMBERSHIP_BRONZE = 'B'
    MEMBERSHIP_SILVER = 'S'
    MEMBERSHIP_GOLD = 'G'
    MEMBERSHIP_CHOICE = [
        (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)
    # this field only takes 3 values either 'B','S' or 'G'
    membership = models.CharField(
        max_length=1, choices=MEMBERSHIP_CHOICE,

    default=MEMBERSHIP_BRONZE)

class Address(models.Model):
    street = models.CharField(max_length=255)
    city = models.CharField(max_length=255)

    #a customer can have multiple addresses
    customer = models.ForeignKey(Customer, on_delete=models.CASCADE)
```

One-to-many relationship of entire data model:

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

```python
class Product(models.Model):
    title = models.CharField(max_length=255)
    description = models.TextField()
    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)

class Customer(models.Model):
    MEMBERSHIP_BRONZE = 'B'
    MEMBERSHIP_SILVER = 'S'
    MEMBERSHIP_GOLD = 'G'
    MEMBERSHIP_CHOICE = [
        (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)
    # this field only takes 3 values either 'B','S' or 'G'
    membership = models.CharField(
        max_length=1, choices=MEMBERSHIP_CHOICE,
default=MEMBERSHIP_BRONZE)

class Order(models.Model):
    PAYMENT_STATUS_PENDING = "P"
    PAYMENT_STATUS_COMPLETED = "C"
    PAYMENT_STATUS_FAILED = "F"
    PAYMENT_STATUS_CHOICE = [
        (PAYMENT_STATUS_PENDING, "Pending"),
        (PAYMENT_STATUS_COMPLETED, "Completed"),
        (PAYMENT_STATUS_FAILED, "Failed")
    ]
    placed_at = models.DateTimeField(auto_now_add=True)
    payment_status = models.CharField(
        max_length=1, choices=PAYMENT_STATUS_CHOICE,
default=PAYMENT_STATUS_PENDING)
    customer = models.ForeignKey(Customer, on_delete=models.PROTECT)

class OrderItem(models.Model):
    order = models.ForeignKey(Order, on_delete=models.PROTECT)
    product = models.ForeignKey(Product, on_delete=models.PROTECT)
    quantity = models.PositiveSmallIntegerField()
    unit_price = models.DecimalField(max_digits=6, decimal_places=2)

class Address(models.Model):
    street = models.CharField(max_length=255)
    city = models.CharField(max_length=255)
    # here customer is a foreign key
    customer = models.ForeignKey(Customer, on_delete=models.CASCADE)

class Cart(models.Model):
    created_at = models.DateTimeField(auto_now_add=True)

class CartItem(models.Model):
    cart = models.ForeignKey(Cart, on_delete=models.CASCADE)
    product = models.ForeignKey(Product, on_delete=models.CASCADE)
    quantity = models.PositiveSmallIntegerField()
```

Many-to-Many relationship:

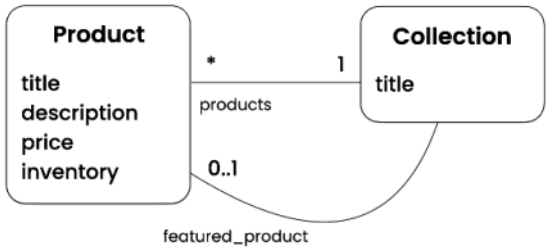
promotion(*)----------------products(*)

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

class Product(models.Model):
    title = models.CharField(max_length=255)
    description = models.TextField()
    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)
```

Resolving Circular Relationships(circular dependency):

When two classes depend on each other. Circular dependency normally happens in this relation.
e.g:
like products depend on collection class and collection depend on products class.

Circular dependency happens here when we django creates a relation in the collection class it automatically creates reverse relation in the product class but we've already created a collection column/field.That's why our name clashes with the one django automatically created.



```python
class Collection(models.Model):
    # title = models.CharField(max_length=255)
    featured_product=models.ForeignKey(Product,on_delete=models.SET_NULL)
class Product(models.Model):
    # title = models.CharField(max_length=255)
    # description = models.TextField()
    # 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)
```

Solution:
So if we don't need django reverse relation we can ignore it using **related_name='+'** and also pass the name of table/model in strings.

```python
class Collection(models.Model):
    # title = models.CharField(max_length=255)
    featured_product = models.ForeignKey(
        'Product', on_delete=models.SET_NULL, null=True, related_name='+')

class Product(models.Model):
    # title = models.CharField(max_length=255)
    # description = models.TextField()
    # 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)
```

CIRCULAR DEPENDENCY

Generic Relationships

We can use generic relationship anywhere, so we design our own app having this general functionality.
For example tags app is created and we can use its ability to tag anywhere.
Django provide **"ContentType" model** that is specifically build to specify generic relations.

For this we need 3 things:
- content_type(to find the table)
- object_id(to find the row in the table)
- content_object(to find the actual object)

```python
from django.contrib.contenttypes.models import ContentType
from django.contrib.contenttypes.fields import GenericForeignKey
# Create your models here

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

class TaggedItem(models.Model):
    # what tag is applied to what object
    tag = models.ForeignKey(Tag, on_delete=models.CASCADE)

    # Type (product,video,article)-using this we can find the table
    # ID - using this we can find the row/record
     #ContentType is a model just like our own models/tables
    content_type=models.ForeignKey(ContentType,
      on_delete=models.CASCADE)

    object_id=models.PositiveIntegerField()
    #now to get the actual product which is tagged
    content_object=GenericForeignKey()
```

section.3.Setting Up the Database

Creating Migrations
Django will look to all the installed apps in our project and make models of that apps. And if you make some changes or update some fields django will detect that and create migrations of that new update.

Migrations files are stored in the ./app/migrations/file.py directory.

```
python manage.py makemigrations
```

We've created a new field "slug" django will detect it and create a new migrations file that will create this field.

```python
class Product(models.Model):
    # title = models.CharField(max_length=255)
    slug=models.SlugField(default='-')

    # 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)
```

Actual sql code:
To see the actual sql code django create while migrating use below command.The generated sql depend on the **type of backend/database** now it is sqlite but for Mysql the generated sql would be different.

```
BEGIN;
--
-- Add field slug to product
--
CREATE TABLE "new__store_product" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
```

```
#here 0003 is the sequence number of the migration.
python manage.py sqlmigrate store 0003
```

```
"slug" varchar(50) NOT NULL, "title" varchar(255) NOT NULL, "description" text NOT
NULL, "inventory" integer NOT NULL, "last_update" datetime NOT NULL, "collection_id"
bigint NOT NULL REFERENCES "store_collection" ("id") DEFERRABLE INITIALLY DEFERRED,
"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;
```

Customizing the database schema:
Advice: Solve simple problem make migration and then solve the next problem.Don't make a lot of changes and make migrations.

Using the Meta class inside the model class you can set the different changes to the model.

```
class Customer(models.Model):
    # MEMBERSHIP_BRONZE = 'B'
    # MEMBERSHIP_SILVER = 'S'
    # MEMBERSHIP_GOLD = 'G'
    # MEMBERSHIP_CHOICE = [
    #     (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)
    # # this field only takes 3 values either 'B','S' or 'G'
    # membership = models.CharField(
    #     max_length=1, choices=MEMBERSHIP_CHOICE,
default=MEMBERSHIP_BRONZE)
    class Meta:
        db_table = 'store_customers'
        indexes = [models.Index(fields=['last_name', 'first_name'])]
```

Reverting migrations:
This will migrate to a specific database state in the past.
But the code exists there.So it is recommended to use git version control.

```
python manage.py migrate <app> 0003
```

←[36;1mOperations to perform:←[0m
←[1m Target specific migration: ←[0m0003_product_slug, from store
←[36;1mRunning migrations:←[0m
 Rendering model states...←[32;1m DONE←[0m
 Unapplying store.0005_auto_20220213_1151...←[32;1m OK←[0m
 Unapplying store.0004_create_zip_field_in_addressClass...←[32;1m OK←[0m

Connecting mysql:
-u:username
-p:database

```
mysql -u root -p storefront
```

Using mysql in django:
Go to our project "storefront\settings.py" path and change it according to your database.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'storefront',
        'HOST':'localhost',
        'USER':'root',
        'PASSWORD':'12345678'
    }
}
```

Running custom query:

First create empty migration file.

```
python manage.py makemigrations store --empty
```

Then run sql query in the operations and two sql queries because the second one is used when you want to revert the changes made in the first query.

```
class Migration(migrations.Migration):
    # dependencies = [
    #     ('store', '0006_reverting _back'),
    # ]

    operations = [migrations.RunSQL("""
    insert into store_collection (title) values ('collection1')
    """, """delete from store_collection where title=('collection1') """)
    ]
```

←[36;1mOperations to perform:←[0m
←[1m Apply all migrations: ←[0madmin, auth, contenttypes, likes, store, tags
←[36;1mRunning migrations:←[0m
 Applying store.0007_auto_20220213_1721...←[32;1m OK←[0m

←[36;1mOperations to perform:←[0m
←[1m Target specific migration: ←[0m0006_reverting _back, from store
←[36;1mRunning migrations:←[0m
 Rendering model states...←[32;1m DONE←[0m
 Unapplying store.0007_auto_20220213_1721...←[32;1m OK←[0m

section.4.Django ORM

Managers and Query sets:

Managers:
Every attribute has a manager which is like interface of a database which provide many function forquerying and updating data.

Executed 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`
```

Query Sets:
Most of the outputs of the managers are called query set. This query set is not evaluted yet but evaluated when you extract some values from it, that's why it is called a lazy evaluation.You can make complex queries using and extract the results using manager functions.

Here **object** is a manager having some functions that forms a query set. Its not necessary that the result of manager is always a query set for instance:
Product.objects.count() gives a single integer.

```
def say_hello(request):

    query_set=Product.objects.all()
    query_set[0]
    return render(request, 'hello.html', {'name': 'Django'})
```

```
FROM `store_product`
LIMIT 1
```

Retrieving objects:

```
#gives query set having all objects
Product.objects.all()

#gives the element having primary key equal to 1.
Product.objects.get(pk=1)

# gives the filtered query set and using first() get the first element.
Product.objects.filter(pk=0).first()

# gives the filtered query set and using first() get the first element
then check if element exists or not.
Product.objects.filter(pk=0).first().exists()
```

Filtering objects:

Below are the django lookup types.

```
from store.models import Product

#greater than
Product.objects.filter(unit_price__gt=10)

#greater than or equal to
Product.objects.filter(unit_price__gte=10)

#less than
Product.objects.filter(unit_price__lt=10)

#less than or equal to
Product.objects.filter(unit_price__lte=10)

#get the range of unit_price using __range
Product.objects.filter(unit_price__range=(20,30))

#get the product containing the word "Coffee" this is case sensitive
product = Product.objects.filter(title__contains="Coffee")

#this is case insensitive
product = Product.objects.filter(title__icontains="coffee")

# gives the titles starting with case insensitive "coffee"
Product.objects.filter(title__istartswith="coffee")

# gives the titles ending with case insensitive "coffee"
Product.objects.filter(title__iendswith="coffee")

#get the products having year 2021
Product.objects.filter(last_update__year=2021)

#get the prducts having description null
Product.objects.filter(description__isnull=True)

#get the prducts having a specific fields
Product.objects.filter(id__in=[1,2])
```

Complex lookups using Q objects:
Applying multiple filters.

```
# Products: inventory < 10 AND price < 20
Product.objects.filter(inventory__lt=10).filter(unit_price__lt=20)

#for OR you've to use Q objects.
from django.db.models import Q

# Products: inventory < 10 OR price < 20
Product.objects.filter(Q(inventory__lt=10) | Q(unit_price__lt=20))

# Products: inventory < 10 OR price not less than 20 "~" sign negates the
condition.
Product.objects.filter(Q(inventory__lt=10) | ~Q(unit_price__lt=20))

Product.objects.filter(Q(inventory__lt=10) & ~Q(unit_price__lt=20))
```

```
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`.`inventory` < 10 AND `store_product`.`unit_price` < 10)
-----------------------------------------------------------------------------------------
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`.`inventory` < 10 OR `store_product`.`unit_price` < 20)
```

```
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`.`inventory` < 10 OR NOT (`store_product`.`unit_price` < 20))
```

```
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`.`inventory` < 10 AND NOT (`store_product`.`unit_price` < 20))
```

Referencing Fields using **F** Objects
These are used to access a particular table field.

```
# Products: inventory = price
Product.objects.filter(inventory=F('unit_price'))

# referencing other tables fields
Product.objects.filter(inventory=F('collection__id'))
```

```
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`.`inventory` = `store_product`.`unit_price`
```

Sorting

.order_by() returns a query set.

```
# sort the title in ascending order
product = Product.objects.order_by('title')

# sort the title in descending order
product = Product.objects.order_by('-title')

# first order by ASCENDING "unit_price" and then by DESCENDING "title"
product = Product.objects.order_by('unit_price','-title').all()

# first order by DESCENDING "unit_price" and then by ASCENDING "title"
product = Product.objects.order_by('unit_price','-title').reverse()

# you can also filter and then apply the order_by method
Product.objects.filter(collection__id=1).order_by('unit_price')

# order by gives query set and indexing gives the first object
Product.objects.order_by('unit_price')[0]

# gives the first object in ascending order
Product.objects.earliest("unit_price")

# gives the first object in descending order
Product.objects.latest("unit_price")
```

```
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`
 ORDER BY `store_product`.`title` ASC

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`
 ORDER BY `store_product`.`title` DESC
```

Limiting

```
# gives the first 5 products: 0,1,2,3,4
Product.objects.all()[:5]

# gives the first 5 products: 5,6,7,8,9
Product.objects.all()[5:10]
```

```
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`
LIMIT 5

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`
LIMIT 5
OFFSET 5
```

Selecting Fields to query

Selecting required columns/fields.
.values() gives dictionary objects directly.

```
# you get dictionary objects of selected fields from the table and also
from other tables
Product.objects.values('id','title','collection__title')
```

```
SELECT `store_product`.`id`,
    `store_product`.`title`,
    `store_collection`.`title`
 FROM `store_product`
INNER JOIN `store_collection`
  ON (`store_product`.`collection_id` = `store_collection`.`id`)
```

```
# you get tuples objects of selected fields
Product.objects.values_list('id','title','collection__title')

# get the "product_id" field in OrderItem create if not exits
OrderItem.objects.values('product_id')

# remove the duplicates
OrderItem.objects.values('product_id').distinct()

# get the ordered products in asdescending order
Product.objects.filter(id__in=OrderItem.objects.values('product_id').distinct()).order_by('title')
```

Deferring objects

Selecting specific fields like values.
.only() gives objects of the model class.

Note: This method can generate a lot of queries. If you iterate on the selected fields.

```
# gives product class objects with 'id' and 'title'
Product.objects.only("id","title")

# gives product class objects with 'id' and 'title'
Product.objects.defer("description")
```

Selecting related fields

Sometimes we need to pre-load some objects **together**.For this we need to tell django to load that tables, django will not look for that tables automatically.

```
# django will only query "Product" table, it will not look other tables unless we tell it to do.For this we use below two methods.
Product.objects.all()

(hello.html)
<li>{{product.title}} - {{ product.collection.title}}</li>

#select_related (1)
#this method pre-load all the related data and joins the two tables.
#use this when other end of the relation have one object.
#iteration on the objects doesn't generate extra queries.
Product.objects.select_related("collection").all()

# prefetch_related (n)
#use this when other end of the relation have many objects.
# but dont iterate on these objects it will generate extra queries.
Product.objects.prefetch_related('promotions').all()

#loading promotions table then loading collection table
Product.objects.prefetch_related('promotions').select_related('collection').all()

# get the last 5 orders with their customer and items(incl product)
# "order" class don't have a items field, so we use reverse relation of
# "orderitem" in the "order" class named "orderitem_set".
# using "orderitem_set__product" we can span to the product table.
Order.objects.select_related('customer').prefetch_related(
        'orderitem_set__product').order_by('-placed_at')[:5]
```

Aggregating objects
For using mathtematical operators.

```
# count the number of products
Product.objects.aggregate(count=Count("id"))

# count and give the min entry in the unit_price field
Product.objects.aggregate(count=Count("id"),min_price=Min("unit_price"))

# apply aggregate on the specific filtered fields
Product.objects.filter(unit_price__gt=
10).aggregate(count=Count("id"),min_price=Min("unit_price"))
```

Annotating objects
Sometimes we need to add some additional attributes to our objects while querying them.

```
# for creating new field
#is_new expects an expression, Value() returns expression thats why we used it.
Customer.objects.annotate(is_new=Value(True))

# create new field "new_id" with values as of primary key "id" and add 1 to each id.
Customer.objects.annotate(new_id=F("id")+1)
```

Calling database functions
Functions are also expressions, so they can be used and combined with other expressions like aggregate functions.

```
from django.db.models import Q, F, Value, Func
```

```python
from django.db.models.functions import Concat

# create new field and use django Func() to concat two fields.
Customer.objects.annotate(
    # CONCAT
    full_name=Func(F('first_name'), Value(" "), F('last_name'),
function='CONCAT'))

# another method
Customer.objects.annotate(
        full_name=Concat('first_name',Value(' '),'last_name'))
```

Grouping data

```python
# no. of each customer orders
Customer.objects.annotate(orders_count=Count('order'))
```

```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`,
    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
LIMIT 21
```

Working with expression wrappers
Django expressions:

1. **Value**:For boolean,number,string.
2. **F**:For referencing fields.
3. **Func**:For database functions.
4. **Aggregate**:For count,Min,Max,etc.
5. **Expression Wrapper**:For complex expressions.

```python
# create expression and then annotate
discounted_price = ExpressionWrapper(F("unit_price")*0.8,
output_field=DecimalField())
queryset = Product.objects.annotate(discounted_price=discounted_price)
```

```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`,
    (`store_product`.`unit_price` * 0.8e0) AS `discounted_price`
FROM `store_product`
LIMIT 21
```

Querying generic relationships

Use the generic models content_type for querying.

Our tags app is decoupled from other apps.In our example we've tags app that is generic and can be used for querying using following method.

```python
# getting tags for a given product
# get the content_type id for the product table
# select_related will pre load the tag table to get rid of extra queries.
# get the filtered tags using the content_type object
content_type=ContentType.objects.get_for_model(Product)
queryset=TaggedItem.objects.select_related("tag").filter(content_type=content_
type, object_id=1)
```

Custom Managers

The above method can be implemented in a more efficient way if we make its class and use its methods.

```python
(models.py)
# create your custom manager class.
class TaggedItemManager(models.Manager):
    def get_tags_for(self, obj_type, obj_id):
        content_type = ContentType.objects.get_for_model(obj_type)
        return
    TaggedItem.objects.select_related("tag").filter(content_type=content_type,
object_id=obj_id)

# create object of the manager in the "TaggedItem" model.
class TaggedItem(models.Model):
    objects=TaggedItemManager()
    tag = models.ForeignKey(Tag, on_delete=models.CASCADE)
    content_type = models.ForeignKey(ContentType, on_delete=models.CASCADE)
    object_id = models.PositiveIntegerField()
    content_object = GenericForeignKey()

(view.py)
# use the custom created method
def say_hello(request):
    TaggedItem.objects.get_tags_for(Product, 1)

    return render(request, 'hello.html', {'name': 'django'})
```

Understanding query set cache

This is a great optimization technique.

```python
# getting data from memory is faster than getting from database.
# after getting the data from database django will store the data in memory
called "query set cache."
# Therefore second list() command is faster becasue it read query set from
cache.

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

Creating objects
Insert a record in database.

Below is the most efficient approach to insert record because if you change some columns of models in models.py file it will automatically change here too.

```
# create each column using database objects
collection = Collection()
collection.title = 'Video Games'
collection.featured_product = Product(pk=1)
collection.save()
```

Updating objects

```
# get all data in memory from database then update it.
collection = Collection.objects.get(pk=10)
collection.title = 'Adventure Games'
collection.featured_product = Product(pk=10)
collection.save()
```

Be default django has tables fields set to '' .When we set some value or update it sets its value.But here we are not updating title so django uses the default empty strings value for **"title"**. To avoid this use the **.update()** method below.

```
# here we are not updating 'title' so django will set its value to ''
collection = Collection.objects.get(pk=10)
collection.featured_product = Product(pk=10)
collection.save()
```

OR

```
# filter the products that you wanna update
Collection.objects.filter(pk=10).update(title='Adventure Games')
```

Deleting Objects

```
# delete single object
collection=Collection(pk=10)
collection.delete()

# delete multiple objects
Collection.objects.filter(id__gt=5).delete()
```

Transactions

Some changes need to be apply all at once if one fails others should also fail.

```
from django.db import transaction

# method.1
# if you want to make the whole function as a transaction
@transaction.atomic
def say_hello(request):
    order = Order()
    order.customer_id =1
    order.save()
    item = OrderItem()
    item.order = order
    item.product_id = 1
    item.quantity = 1
    item.unit_price =10
    item.save()
    return render(request, 'hello.html', {'name': 'django', 'tags': 1})

# method.2
# if you want some part of function as a transaction
def say_hello(request):
    # some other code
    # blah blah blah

    # transaction code
    with transaction.atomic():
        order = Order()
        order.customer_id =1
        order.save()
        item = OrderItem()
        item.order = order
        item.product_id = 1
        item.quantity = 1
        item.unit_price =10
        item.save()
    return render(request, 'hello.html', {'name': 'django', 'tags': 1})
```

26- Executing Raw SQL Queries

```
from django.db import connection


# method.1
# but we don't have other methods like filter, annotate, etc
# query maps to our model layer
query_set = Product.objects.raw('SELECT * FROM store_product')

# method.2
# sometimes we need queries that dont map to our model objects so we need to bypass the model layer
#  using following method
def say_hello(request):
    with connection.cursor() as cursor:
        cursor.execute('SELECT * FROM store_product')
```

```
    return render(request, 'hello.html', {'name': 'django', 'tags': 1})

# method.3
# another method is to encapsulate the sql query in a stored procedure and
call that here
# "say_hello" is the procedure with 'Hello, world!' as parameters.
with connection.cursor() as cursor:
    cursor.callproc('say_hello', ['Hello, world!'])
```