**Model in django**

* A model is a class that represents table or collection in our DB, and where every attribute of the class is a field of the table or collection. Models are defined in the app/models.py
* Each model is a Python class that subclasses [django.db.models.Model](https://docs.djangoproject.com/en/3.1/ref/models/instances/" \l "django.db.models.Model" \o "django.db.models.Model).
* Each attribute of the model represents a database field

With all of this, Django gives you an automatically-generated database-access

Lets take a n exapme



The above Person model would create a database table like this:



**Model Meta options**

We will see some possible [metadata options](https://docs.djangoproject.com/en/dev/topics/db/models/#meta-options) that you can give your model in its internal classMeta.

Meta class is subclass in mode name/method of model./py file

1. **db\_table=<table\_name>**

The name of the database table to use for the model. By default django creates table by name <appsName\_modelname>

1. **app\_label=<myapp>**

If a model is defined outside of an application in [INSTALLED\_APPS](https://docs.djangoproject.com/en/dev/ref/settings/#std:setting-INSTALLED_APPS), it must declare which app it belongs to

1. **options.permissions/models.permissions=[permissions]**

Extra permissions to enter into the permissions table when creating this object. Add, change, delete, and view permissions are automatically created for each model

permissions=[('can\_deliver\_pizzas','Can deliver pizzas')]

This is a list or tuple of 2-tuples in the format (permission\_code,human\_readable\_permission\_name).

1. **Options.default\_permissions<permissions>**
2. **Options.constraints=<list of constrainsts>**

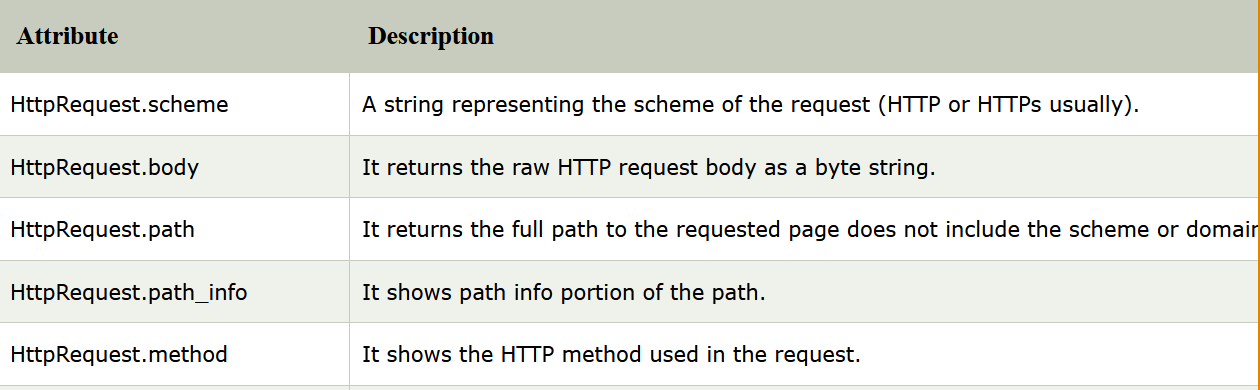
A list of constraints that you want to define on the model

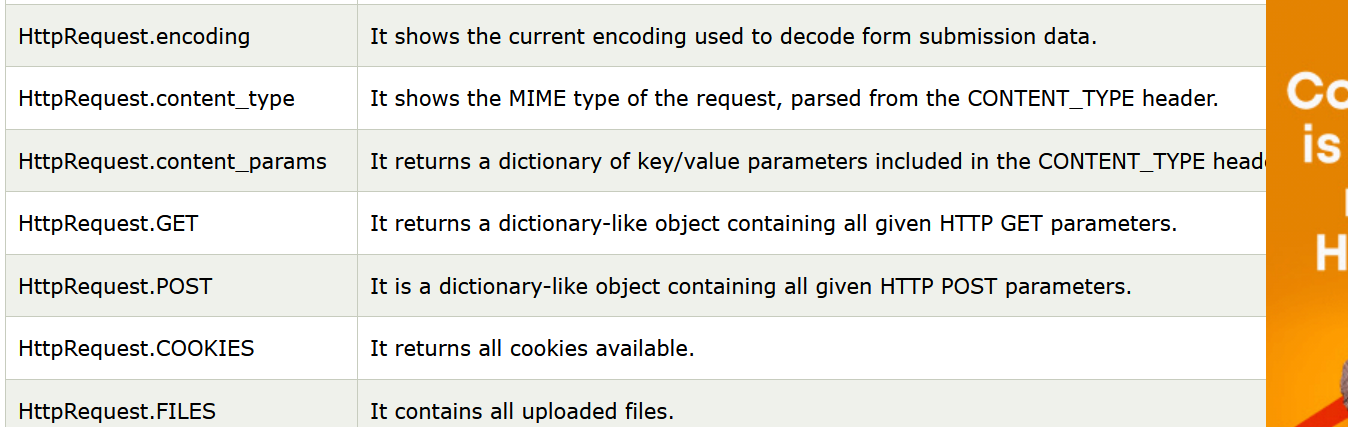
1. **abstract=True** ----- In model interitance
2. **proxy=True** ---- In model inheritance for Proxy Modal inheritance

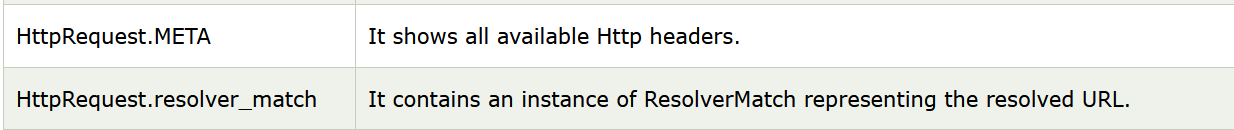
**Adding data to DB**



**HttpRequest in Django**







*request.body ---- it’s typical use in in REST api while getting data from request.*

**QuerySets**

A djangoqueryset is like its name says, basically a collection of (sql) queries, in your example above print(b.query) will show you the sql query generated from your django filter calls

Since querysets are lazy, the database operation isn't done immediately but operations happens only when needed or when the queryset is evaluated

**QuerySets are lazy**

QuerySets are lazy – the act of creating a QuerySet doesn’t involve any database activity. You can stack filters together all day long, and Django won’t actually run the query until the QuerySet is evaluated.

**>>>**q=Entry.objects.filter(headline\_\_startswith="What")

**>>>**q=q.filter(pub\_date\_\_lte=datetime.date.today())

**>>>**q=q.exclude(body\_text\_\_icontains="food")

**>>>**print(q)----At this line DB will be hitted,there are many easy to evaluequerysets

**Evaluating QuerySets**

* Internally, a QuerySet can be constructed, filtered, sliced, and generally passed around without actually hitting the database. No database activity actually occurs until you do something to evaluate the queryset.
* We can evaluate query in following ways

1. **Iteration**

A QuerySet is iterable, and it executes its database query the first time you iterate over it. For example, this will print the headline of all entries in the database



1. **repr()**

A QuerySet is evaluated when you call repr() on it. This is for convenience in the Python interactive interpreter, so you can immediately see your results when using the API interactively.

1. **len()**

A QuerySet is evaluated when you call len() on it.

*It returns the length of the result list ,*For this purpose django provides a **count()** method for precisely this reason.

1. **list()**

Force evaluation of a QuerySet by calling list() on it. For example:



1. **bool()**

Testing a QuerySet in a boolean context, such as using **bool(), or, and or an if** statement, will cause the query to be executed

If there is at least one result, the QuerySet is True, otherwise False.

1. **pickling/caching**
2. **slicing**

* If you use index to pick one/more element from a not evaluated QuerySet, it causes database hits
* you pick from an already evaluated QuerySet it uses cache(from stored variable or result)
* Slicing a not-evaluated QuerySet returns a new QuerySet. The returned QuerySet does not allow further modifications (e.g., adding more filters, or modifying ordering) but it does allow more slicing. Queryset (either sliced or not) saves results to its cache if you iterate over it



Note:

We can do chaining on queryset.

Not all methods from queryset API return a new query set when applied on a queryset

**Methods that return new QuerySets**

<https://docs.djangoproject.com/en/3.1/ref/models/querysets/#when-querysets-are-evaluated>

-------------- Methods which return a new query set

There are many methods for this but here we will see few of them

1. **filter(\*\*kwargs)/filter()**

Returns a new QuerySet containing objects that match the given lookup parameters.

If you need to execute more complex queries (for example, queries with OR statements), you can use Q objects.

**Modelname.objects.filter(db\_column\_name=val)**

**db\_column\_name** --- this is column name or file name from model class

1. **exclude()/exclude(\*\*kwargs)**

Returns a new QuerySet containing objects that do not match the given lookup parameters/conditions

1. **order\_by(col\_name)/order\_by(\*field)**

By default, results returned by a QuerySet are ordered by the ordering tuple given by the ordering option in the model’s Meta.

1. **col\_name**---- then orders in ascending order
2. -**col\_name**----- then order/sorts in descending order
3. ?**----** random order

You can override this on a per-QuerySet basis by using the order\_by method



***students.objects.order\_by(‘-rollNumber’)*** ---- orders in descending order of roll number

***students.objects.order\_by(‘?’)* -----** randon order(for any column**)**

1. **reverse()**

Use the reverse() method to reverse the order in which a queryset’s elements are returned.

Calling reverse() a second time restores the ordering back to the normal direction.



To retrieve last 5 items in querysets

***student.objects.order\_by(‘id’).reverse()[:5]*** ---- last 5 rows in reverse order of roll number

1. **all()** ---fetched all data for given table or model
2. **values(column\_name(s))/values()**

This is used to fetch required column data in dictionary form. Column name will be key for each queryset objects

If no column name is given it behaves same as all()

If column name or names (comma seperated) are given then it fetches those required column data

1. **values\_list(column\_names,flat=False,names=False)**

Returns values as tuple for given column names.

If no field/column name is passed then returns all data

If fiels/couln names are given then it returns those column data

If name=True then retuns data as named tupple

1. **dates(field, kind, order='ASC')**

Returns queryset that returns to datetime.date object

field---🡪 this is name of datafield of your model name or column name

kind --🡪 it should be either year, month, week or day

order 🡪 ASC or DESC

***student.object.date(‘PASS\_YEAR,year,ASC)***

1. **datetime(field, kind, order='ASC')**
2. **none()** ---------- don’t retuns anything. Probably it’s used is to evaluating the query.
3. **qs1.union(\*other\_queryset,all=FALSE)**

Used to perform sql union. By default it returns unique result after SQL union operation.

**It required that both queryset should have same column name and data types else error.**

All=TRUE --- then it returns duplicate results also after union operation

1. **qs1.intersect(qs2,all=False)**

uses SQL's INTERSECT operator to retun the shared elements of two or more queryset.

e.g.— ***qs1.intersection(qs2, qs3)***

1. **qs1.difference(\*other\_qs)**

uses SQL's EXCEPT operator to return the shared queryset.

1. **get(some\_condition(s))/get()**

it returns data not queryset based on given condition(applied on column name or something else)

e.g:-

Entry.objects.get(blog=blog, entry\_number=1)

Entry.objects.get(pk =1)

**Operations which return a new queryset**

**AND(&)**and **OR** operation returns a new queryset

1. **AND**

Combines two QuerySets using the SQL AND operator.

**qs1 AND qs2**

**filter(condition1,condition2)**

**filter(Q(condition1)&Q(condition2))**



Q(xyz)----- this is Q class we will see it later in this notes

This querysetevaluates to



1. **OR**

Combines two QuerySets using the SQL OR operator.

**qs1 OR qs2**

**qs1|qs2**

**filter(Q(condition1)|Q(condition2))**



Above queryset evaluates to



1. **NOT**

**exclude(condition)**

**filter(~Q(condition))**

**Methods that do not return QuerySet**

<https://docs.djangoproject.com/en/3.1/ref/models/querysets/#methods-that-do-not-return-querysets>

------------ Methods which don’t return a new queryset

There are many methods which don’t return queryset , we will see some of them

1. **get()/get(\*\*kwargs)**

Returns the object matching the given lookup parameters, which should be in the format described in Field lookups.



**NOTE**:

1. If get() finds more than one object, it raises a Model.MultipleObjectsReturned exception.
2. If get() doesn’t find any object, it raises a Model.DoesNotExist except

We can handle those exceptions using "**django.core.exceptions**"

1. **2. create()/create(\*\*kwargs)**

A convenience method for creating an object and saving it all in one step. It returns created object.

**Example:**

Employee.objects.create(eno=201,ename='Sunny',esal=30000,eadd='MUM')

***Question***

Save/add data to student table(already created)

***Using save()***

S=Student(name=’myname’ , roll= 122334)

s.save()

***Using create***

Student.create(name=’myname’ , roll= 122334)

1. **get\_or\_create()/get\_or\_create(\*kwargs)**

A convenience method for looking up an object with the given kwargs (may be empty if your model has defaults for all fields), creating one if necessary.

1. **update / obj.update(fd1=value1, fd2=value2 ..)**

It is used to update objects (On which it is called) in DB

**Question**

update the name and roll for student whose id is 100.

**Solution**

Student.object.filter(id=100).update(name=myname, roll=12233)

1. **update\_or\_create(defaults=None, \*\*kwargs)/update\_or\_create()**

If record exists with given attribute/parameter then update the value for that object with default value else create a new record with given value.

Let us take example of create or update manually.

**obj, created = Person.objects.update\_or\_create(**

**first\_name='John', last\_name='Lennon',**

**defaults={'first\_name': 'Bob'},**

**)**

*# If person exists with first\_name='John' & last\_name='Lennon' then update first\_name='Bob'*

*# Else create new person with first\_name='Bob' & last\_name='Lennon'*

A convenience method for updating an object with the given kwargs, creating a new one if necessary. The defaults is a dictionary of (field, value) pairs used to update the object. The values in defaults can be callables.***queryset.update(col\_name=val1, col\_name2= val2…)***

e.g. ***Student.object.filter(id=100).update(name=myname, roll=12233)***

Returns a tuple of (object, created), where object is the created or updated object and created is a boolean specifying whether a new object was created.

Above can be written easily using create\_or\_update() as below



1. **bulk\_update()/bulk\_update([enrty1,entry2…entryn] fields, batch\_size=None)**--- objs are already created queryset
2. **count()**
3. **in\_bulk()/in\_bulk(id\_list=None, field\_name='pk')**
4. **latest(\*arg)**---- returns the latest object based on spplied field. e.g – Mobile.object.latest(‘released\_date)
5. **earliest((\*arg)** --- Same as above but for earliest
6. **latest(\*fields)**
7. **first()**-- returns the first object of query, if ordering is not defined then it orders on primary key
8. **last()** --- returns the last object of query, if ordering is not defined then it orders on primary kev
9. **bulk\_create()** --- this is used for creating/saving many objects in DB at a time.

Entry.objects.bulk\_create([enrty1,entry2….entryn])----- Syntax

Example:

emp1=Employee(eno=11,ename=’Sunny’,esal=30000,eadd=’MUM’)

emp2=Employee(eno=11,ename=’Sunny’,esal=30000,eadd=’MUM’)

emp3=Employee(eno=11,ename=’Sunny’,esal=30000,eadd=’MUM’)

Employee.objects.bulk\_create([emp1,emp2,emp3])

1. **bulk\_create(objs, batch\_size=None, ignore\_conflicts=False)**
2. **aggregate(\*args, \*\*kwargs)** ----Returns a dictionary of aggregate values (averages, sums, etc.) calculated over the QuerySet. Used for SQL aggregate function (AVG, SUM, MIN, MAX, COUNT)

**Example**--

avg=Employee.objects.all().aggregate(Avg('esal'))

1. **delete**  ----- For deleting one/more from DB. If called on queryset then delete more than on record.

object\_name.delete() ------- SYntax

emp=Employee.objects.get(id=200)

emp.delete()

1. **order\_by (field\_name)** ------- Ordering the data on given field in ascending/decending order.

Entry.object.all().order\_by(field\_name) – Ascending order

Entry.object.all().order\_by(-field\_name) ---- Descending order, if added – with field then descending order.

**Field lookups**

Field lookups are how you specify the **meet of an SQL WHERE clause**. They’re specified as keyword arguments to the QuerySet methods filter(), exclude() and get().

Django’s built-in lookups are listed below.:-

Queryset\_for\_getting\_data(colName\_\_lookuptype=val)----------Syntax to use(careful about double underscore)

**Sample queries**

*Entry.objects.filter(colName\_\_lookuptype=value)* ----------Syntax to use

*Entry.objects.get(colName\_\_lookuptype=value)* ----

e.g ---- Studen.objects.filter(marks\_\_lt=30)

1. **exact**

Exact match. If the value provided for comparison is None, it will be interpreted as an SQL NULL (see [isnull](https://docs.djangoproject.com/en/3.1/ref/models/querysets/#std:fieldlookup-isnull) for more details).

Example:-



1. **iexact**

Case-insensitive exact match. If the value provided for comparison is None, it will be interpreted as an SQL NULL (see isnullfor more details).

1. **Contains**

Case-sensitive containment test.



1. **Icontains**

Case-insensitive containment test.

1. **In**
2. **gt**
3. **gte**
4. **lt**
5. **lte**
6. **startswith**
7. **istartswith**
8. **endswith**
9. **iendswith**
10. **range**
11. **date**
12. **year**

**and so on**

**Caching and QuerySets**

Each [QuerySet](https://docs.djangoproject.com/en/3.1/ref/models/querysets/#django.db.models.query.QuerySet) contains a cache to minimize database access.

To enable cache in QuerySet, simply save the QuerySet in a variable and reuse it

Example:-

print([e.headline for e in Entry.objects.all()])

print([e.pub\_date for e in Entry.objects.all()])

Above code will hit DB twice (each time query is executed)

Below code will execute only one and then result will be used from cached query



1. **Field types**

It is datatypes defined in SQL or datatype of each column in DB table.

In djnago we specify the field type/data type of each column of table/mode name in the model name

More we can check on:-

<https://docs.djangoproject.com/en/3.1/ref/models/fields/#field-types>

* 1. **CharField(max\_length=None, \*\*options)**
  2. **DateField(auto\_now=False, auto\_now\_add=False, \*\*options)**
  3. **DateTimeField(auto\_now=False, auto\_now\_add=False, \*\*options)**
  4. **DecimalField(max\_digits=None, decimal\_places=None, \*\*options)**
  5. **mailField(max\_length=254, \*\*options)**
  6. **FileField(upload\_to=None, max\_length=100, \*\*options)**
  7. **ImageField(upload\_to=None, height\_field=None, width\_field=None, max\_length=100, \*\*options)**
  8. **IntegerField(\*\*options)**
  9. **GenericIPAddressField(protocol='both', unpack\_ipv4=False, \*\*options)**

1. **Field Options**

Field option is something like **constrains in SQL**

Few are listed below, All available field options we can check it on below link

<https://docs.djangoproject.com/en/3.1/ref/models/fields/#field-options>

* 1. **Field.null**---If True, Django will store empty values as NULL in the database. Default is False.
  2. **Field.blank**---If True, the field is allowed to be blank. Default is False.
  3. **Field.db\_column**---The name of the database column to use for this field. If this isn’t given, Django will use the field’s name
  4. **Field.default**---default value for rhat field
  5. **primary\_key**---If True, this field is the primary key for the model.

**Field.unique**---If True, this field must be unique throughout the table

**Related\_name** --- this is used to specifies the name of the reverse relation from the one model(which is giving foreign key into another model) back to your model. Let’s take example—

class Post(models.Model):

title = models.CharField(max\_length = 63)

tags = models.ManyToOne(User, related\_name ='blog\_posts')

user.blog\_post --- this will give us all post made by specified user.

**Relationship in Django**

We learn relationship in Django. It has three type

1. Many-to-One
2. Many-to-Many
3. One to One

In django foreign\_key column is created by name PrimaryKeyOfAnotherTable\_PrimaryKeyOfThisTable

1. **OnetoOne relatioship**

It is used when there is need to maintain one to one relationship, i.e. one record in a table can relate to one record in another table.

e.g—let say we have two table, User and Page, One user can create only One page on social media.

Class Model1(models.model):

fd1

fd2

class Model2(models.Model):

fdx

fdy

model1=models.OneToOne(Model1,on\_delete=models.CASCADE)

**NOTE:**

* Foreign key column name will be PrimaryKeyOfModel1\_PriamryKeyOfModel2 in table2/model2
* If required then, foreign key (model1 in Model2) can be as primary key b/c relation is OneToOne.
* on\_delete is optional

Example:

We have two model -- Page model and User model( in built so not required to write it) and want to establish OneToOne b/w them.

Models.py file

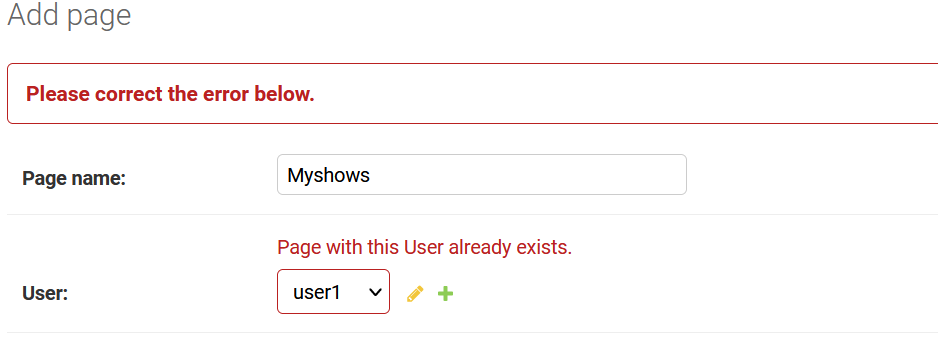
class Page(models.Model):

    page\_name=models.CharField(max\_length=10,primary\_key=True)

    user=models.OneToOneField(**User**, on\_delete=models.CASCADE)

Result:

Let say we have one record in which 'user1' is already added for Page table if we try to add 'user1' in another table then we will get below error b/c one to one relationship.



1. **Many-to-One relationship**

For managing many to one relationship we use **models.ForeighKey(modelname, on\_delete)** to acheive it.

This is used in the model/table which hold the foreign key

**Creating OnetoMany relationship between Customer and Order model**

class Customer(models.Model):

    name=models.CharField(max\_length=20, primary\_key=True)

class Order(models.Model):

    customer=models.ForeignKey(Customer, on\_delete=models.CASCADE)

    order\_no=models.CharField(max\_length=200)

**models.ForeighKey(modelname, on\_delete)/models.ForeignKey(to, on\_delete, \*\*options)**

this is class in models API

A many-to-one relationship. Requires two positional arguments: the **class/object of class** to which the model is related and the on\_delete option.

*Arguments of ForeignKey()*

* **on\_delete argument/on\_delete=?**

When an object referenced by a ForeignKey is deleted, Django will emulate the behavior of the SQL constraint specified by the on\_delete argument. Available options:-

* + - CASCADE
    - PROTECT
    - RESTRICT
    - SET\_NULL-- Set the ForeignKey null; this is only possible if null is True
    - SET\_DEFAULT ---- Set the ForeignKey to its default value; a default for the ForeignKey must be set.
    - DO\_NOTHING --- Take no action

1. **Many-to-many relationships/ManyToManyField(to,\*\*options)**

* To define a many-to-many relationship, use ManyToManyField
* To establish many to many relationship python creates intermediate table name by tablele1\_table2(by default), we can change the intermediate table name also.
* A many-to-many relationship. Requires a positional argument: the class to which the model is related, which works exactly the same as it does for ForeignKey, including recursive and lazy relationships.
* Related objects can be added, removed, or created with the field’s RelatedManager.

**Arguments of ManyToMany field**

* **ManyToManyField.through**
  + - * If manually specify the intermediary table
      * The most common use for this option is when you want to associate extra data with a many-to-many relationship

Example:-

Below snip of code will have intermediate table by name "Membership" not Person\_Group(by default)





**ManyToManyField.db\_table**

The name of the intermediatory table to create for storing the many-to-many data

By default, Django will create **table1\_table2** name as intermediatory table

**Generalize way of declaring model**

From all above discussion we can generalize declarations of modelname in below way.

*class Modelname(model.Model):*

*table\_column\_name = models.Filed\_Type(Filed\_Options)*

*table\_column\_name = models.Filed\_Type(Relationship\_fields, Fiel\_Options)*

##############################################################################################

# Getting data of foreign key table from primary key field table object --- backward relation #

###############################################################################

We can do it without using **‘related\_field’** attribute and with ‘related\_field’ attribute defined in out foreign key field.

**CASE 1: Without related\_feild**

We will have to use objectname.foreignkeytable\_set variable

class Skill(models.Model):

    name = models.CharField(max\_length=256)

class Subskill(models.Model):

    skill = models.ForeignKey(Skill, on\_delete=models.CASCADE)

    sub\_name = models.CharField(max\_length=256)

Note: Here Subskill is working as foreign key field table and skill as primary key table

Views.py file

def testrel(reqeust):

    skill\_obj = Skill.objects.all()[0]

    print(skill\_obj.name)

    subskills = skill\_obj.subskill\_set.all() # now we have data of subskill table

    print('subskills data are:')

    for each in subskills:

        print(each.sub\_name)

    return HttpResponse('requried data retrived')

**CASE 2: Witht related\_field defined in foreign\_key table**

class Skill(models.Model):

    name = models.CharField(max\_length=256)

class Subskill(models.Model):

    skill = models.ForeignKey(Skill, on\_delete=models.CASCADE,related\_name='myrelation')

    sub\_name = models.CharField(max\_length=256)

views.py file

def testrel(reqeust):

    skill\_obj = Skill.objects.all()[0]

    print(skill\_obj.name)

    subskills = skill\_obj.myrelation.all()

    print('subskills data are:')

    for each in subskills:

        print(each.sub\_name)

    return HttpResponse('requried data retrived')

####################################

# select\_related INNER JOIN#

####################################

Actually, it’s used for performing 'INNER JOIN'.

Returns a QuerySet that will “follow” foreign-key relationships, selecting additional related-object data when it executes its query.

Example: let take below two models.

*class Product(models.Model):*

*name = models.CharField(max\_length=50)*

*product\_rank = models.OneToOneField('ProductRank') # can be fofeignkey also*

*class ProductRank(models.Model):*

*rank = models.IntegerField(default=0)*

**Query1:**

Product.objects.filter(name=’mobile’).select\_related()

If we run the above query then it will fetch product data and related data of ProductRank data

**Note:**

The order of filter() and select\_related() chaining isn’t important. These querysets are equivalent:

Entry.objects.filter(pub\_date\_\_gt=timezone.now()).select\_related('blog')

Entry.objects.select\_related('blog').filter(pub\_date\_\_gt=timezone.now())

**Advantage:**

It **reduces the number of database hit** by query or django, let take below case

**Without select\_related**

e = Entry.objects.get(id=5) # Hits the database.

b = e.blog # Hits the database again to get the related Blog object.

**With select\_related**

e = Entry.objects.select\_related('blog').get(id=5) # required data in single hit

####################################

# select\_related OUTER JOIN#

####################################

We can do left join using select\_related also. We need to just use all() instead of filter().

**models class**

*class Product(models.Model):*

*name = models.CharField(max\_length=50)*

*product\_rank = models.OneToOneField('ProductRank') # can be fofeignkey also*

*class ProductRank(models.Model):*

*rank = models.IntegerField(default=0)*

**Django query for outer join**

Product.object.all().select\_related()

**Note:**

The limitation of this query is that it can do left join from ForeignKey table to PrimaryKey table.

##############################

# Model Inheritance #

##############################

There are 4 types of inheritance in django for models-

1. Abstract Base class Model inheritance
2. Multi table inheritance
3. Proxy model inheritance
4. Multilevel inheritance

#################################

#Abstract Base Class Model Inheritance#

#################################

This is used, when we have more than one model and some fields are common in all Model.

Here model for which we have used abstract=True in Meta option will not be created in DB.

Class Model1(models.Model):

#Field1

#Field 2

**Class Meta:**

**abstract=True** **# this means Model1 table will not be created in DB**

Class Model2(Model1)

Name=models.charfield(max\_length=256)

Age=models.IntegerField()

**Note:**

* abstract=True will instruct django to not create table in DB, just for indication that this is for inheritance.
* All fields of Model1 will be inherited to Model2 and table for model1 will not be created.

e.g—Student and Teacher Model

class Student(models.Model):

name=models.CharField(max\_length=256)

email=models.EmailField()

address=models.CharField(max\_length=300)

roll=models.IntegerField()

course=models.CharField(max\_length=100)

class Teacher(models.Model):

name=models.CharField(max\_length=256)

email=models.EmailField()

address=models.CharField(max\_length=300)

teacher\_id=models.IntegerField()

department=models.CharField(max\_length=100)

In both models name,email,address are common so if we use abstract base model inheritance then we can easily define it as below---

class ContactInfo(models.Model):

    name=models.CharField(max\_length=256)

    email=models.EmailField()

    address=models.CharField(max\_length=300)

class Meta:

abstract=True # now Contact Info table will not be created in DB

class Student(ContactInfo):

    roll=models.IntegerField()

    course=models.CharField(max\_length=100)

class Teacher(Student):

    teacher\_id=models.IntegerField()

    department=models.CharField(max\_length=100)

#################################

# Multi Table Inheritance #

#################################

This is same as Abstract Base Model Inheritance the difference is that it Base table will be also created.

Class Model1(models.Model):

#Field1

#Field 2

Class Model2(Model1)

Name=models.charfield(max\_length=256)

Age=models.IntegerField()

Note:

* Here both model1 and Model2 will be created b/c we have not used abstract=True in Model1.
* Here parent class is used as fogeign key in child class while checking description of table but when we try to add data in child class then it asks to add all fields.

#################################

# Multi Level Inheritance #

#################################

In such type of inheritance one child class inherits more than one parent class.

Class Model1(models.Model)

fd1

fd2

class Model2(models.Model)

fdx1

fdx2

class Model3(Model1,Model2)

fdy1

fdy2

Note:

* Here all fields of Model1,Model2 (parent class) will be inherited to Model3 (child class)
* All child and parent class will be created into DB.

**Model managers**

It is used for interacting with DB using models.

*A model's manager is an object through which Django models perform database queries.*

Each Django model has at least one manager, and you can create custom managers in order to customise database access.

e.g--

Employee.objects.all()

**Note:**

* Here objects is model manager of Employee Model class.
* Whenever we call **objects** on any model name then it points to default model manager.
* Whenever we call Entry.objects it call default get\_queryset() of model.

type(Employee.objects) ------ django.db.models.managers.Manager

**Custom Model manager**

We can write custom model manager for any model using below syntax—

Class CustomManager(models.Manager):

def get\_queryset(self): # Overriding default get\_queryset of modal class

return super().get\_queryset()

class MyModal(models.Modal):

fd1

fd2

objects=CustomManager()

* **get\_queryset() is also available in rest framework generic views, used for filtering/getting data using query string.**

**Note:**

* We can use any word instead of 'objects ' but if used then where ever objects is used that need to be replaced by new word.
* Now whenever we call objects on MyModal class then it will point to MyModal class manager not default modal class manager.

**Writing custom methods for fetching/filtering data**

We can write our custom methods in custom manager for fetching/filtering data from DB

**Example:**

Write method to fetch data in ascending order or salary.

Class CustomManager(models.Manager):

def get\_queryset(self):

return super().get\_queryset()

def get\_by\_ascending\_sal(self):

return super().get\_queryset().order\_by('esal ')

class Employee(models.Model):

ename

esal

objects=CustomManager()

##################################

# Proxy model inheritance #

##################################

It is another inheritance type in which at backend only one table will be created but for view purpose we can see two table. One original table one proxy table.

Class MyModel(models.Model):

fd1

fd2

class ProxyMyModel(MyModel): #this is now proxy table

class Meta:

proxy=True

**Note:**

* Here we can see two table ProxyMyModel and MyModel table but at backend only one table 'MyModel' will be present. 'ProxyMyModel' is proxy table.
* If required we can define custom manager for Proxy modal also.

##############################################################

# Overriding in-built model method or creating new modal methods #

##############################################################

We can override model method behavior. In particular you’ll often want to change the way save() and delete() work.

**Method 1:** Creating/overriding in model class

Class MyModel(models.Model):

fd1

fd2

def method(self,\*args,\*\*kwargs):

#do something

**Method 2:** By using custom modal managers

Already done above.

**Overriding save method**

Here we are taking Blog class as Model class.

class Blog(models.Model):

name = models.CharField(max\_length=100)

tagline = models.TextField()

def save(**self, \*args, \*\*kwargs**):

do\_something()

super().save(\*args, \*\*kwargs) # Call the "real" save() method.

do\_something\_else()

**Advantages:**

* We can override/write methods to perform something before and after doing some action by those methods.

**Questions:**

Write logic (override save() ) for case --- if user name length is of even character then he can’t have any blog account or can save data.

**Solution:**

django.core.exceptions import PermissionDenied

class Blog(models.Model):

name = models.CharField(max\_length=100)

tagline = models.TextField()

def save(self, \*args, \*\*kwargs):

if len(self.name)%2=0:

raise PermissionDenied(‘not allowed to save’) # You can’t havev any blog account!

else:

super().save(\*args, \*\*kwargs) # Call the "real" save() method.

**Looping in Django templetes**

We can use for loop in Django templates in below way

*{% for i in list %}*

*#statements*

*{% endfor %}*

**Getting value from Django context**

It is accessed using context variable

{{ }} -----------this is called context variable

1. **Using dot operator**

context\_name.context\_variable\_name

For more check variable and lookup in django template (below on this [age)

e.g:-

{{ item.item\_name }}

{{ item.item\_desc }}

1. **Directly from variable**

*{{ context\_variable\_nmae))*

Let say I have created context as below



Now we can access value for each context variable in below ways



**Variable and lookup in django templates**

<https://docs.djangoproject.com/en/3.1/ref/templates/api/>

Variable names must consist of any letter (A-Z), any digit (0-9), an underscore (but they must not start with an underscore) or a dot.

Dots have a special meaning in template rendering. A dot in a variable name signifies a lookup. Specifically, when the template system encounters a dot in a variable name, it tries the following lookups, in this order:

* Dictionary lookup. Example: foo["bar"]
* Attribute lookup. Example: foo.bar
* List-index lookup. Example: foo[bar]

e.g- {{ var\_name.index }} -------- to access value from a list(index based) variable based on index value

order=[‘rakesh’,’kuchwaha’]

order\_data=[]

for each in order:

            #print(each)

            l=[]

            l.append(each)

            order\_data.append(l)

order\_details={'order\_info':order\_data}

return render(request,'woodshophome/order\_status.html',order\_details)

In template or htmlpage we can access value by below way

{% for row\_data in order\_info %}

                        <tr>

                            <td><a>{{ row\_data.0 }}</td>

                            <td>{{ row\_data.1 }}</td>

                            <td>{{ row\_data.2 }}</td>

                        </tr>

                    {% endfor %}

**Django templates tag**

All available templates can be found on below link

<https://docs.djangoproject.com/en/3.1/ref/templates/builtins/>

Since the template system has no concept of “escaping”, to display one of the bits used in template tags, you must use the {% templatetag %} tag



1. **for**

Loops over each item in an array, making the item available in a context variable.

For example, to display a list of athletes provided in athlete\_list:

<ul>

{% for athlete in athlete\_list %}

<li>{{ athlete.name }}</li>

{% endfor %}

</ul>

1. **if(**{%if%}**)**

The {%if%} tag evaluates a variable, and if that variable is “true” (i.e. exists, is not empty, and is not a false boolean value) the contents of the block are output

{% if athlete\_list %}

Number of athletes: {{ athlete\_list|length }}

{% elif athlete\_in\_locker\_room\_list %}

Athletes should be out of the locker room soon!

{% else %}

No athletes.

{% endif %}

1. **url**

Returns an absolute path reference (a URL without the domain name) matching a given view and optional parameters. Any special characters in the resulting path will be encoded using iri\_to\_uri()

*{% url 'some-url-name' v1 v2 %}*

*{% url 'app\_name:view-name' %}*

*{% url 'some-url-name' arg1=v1 arg2=v2 %}*

**NOTE**:

1. The first argument is a [URL pattern name](https://docs.djangoproject.com/en/3.1/topics/http/urls/#naming-url-patterns). It can be a quoted literal or any other context variable. Additional arguments are optional and should be space-separated values that will be used as arguments in the URL
2. View-name is the name given in the url.py file for setting url, it’s not model’s view name

Let say we have below date in urls.py filefor app “myapp”

path('myapp/', views.detail, name='myapp-detail')

then url should be like:

{% url 'myapp:myapp-detail' %}

1. **extends ({% entends %}**

This used for reusing/extending some sort of templated/templates object

Normally the template name is relative to the template loader’s root directory. A string argument may also be a relative path starting with ./ or ../

* **{% extends "base.html" %}**

Uses the literal value "base.html" as the name of the parent template to extend.

* **{% extends variable %}**

If the variable evaluates to a string, Django will use that string as the name of the parent template

If the variable evaluates to a Template object, Django will use that object as the parent template



* **{% include ‘templateName’ %}/{% include ‘varname’ %}**

This is used to include a part of HTML code into another html page

Extend is used for including the whole layout

**Removing hardcoded/Dynamic Url**

The problem with this hardcoded, tightly-coupled approach is that it becomes challenging to change URLs on projects with a lot of templates

Let’s take example of one hardcoded url”

*<li><a href="/polls/{{ question.id }}/">{{ question.question\_text }}</a></li>*

---- this is hardcoded url as we are pasting link in html page it doesn’t required any modification in urls.py

--- this will work for few question ids url which are hardcoded here

We can remove the hardcoded urliin below way

*<li><a href="{% url 'detail' question.id %}">{{ question.question\_text }}</a></li>*

---- No w this will work for each question id

For above urls.py file should have configured like-

*path('<int:question\_id>/', views.detail, name='detail')*

##################################

# Advance Template Feature #

##################################

1. Template inheritance --------- Use of extends key. Using one template into other.
2. Filters

######################

# Template Filters #

######################

Django Template Engine provides filters which are used to transform the values of variables and tag arguments. It can be classified in two

1. Built-in template filter
2. Custom template filter

{{ variable\_name | filter\_name }} -------- Syntax to use template filter

{{ value | filter\_name }} ---------- Syntax

Example:

{{ value | length }} --- to get length of value

There are many building filter available in django we can get them all on below link

<https://docs.djangoproject.com/en/3.2/ref/templates/builtins/#built-in-filter-reference>

**Example:**

{{ value|add:"2" }}

f **value** is **4**, then the output will be **6**

**Some built-in filter:**

**addslashes**--{{ value|addslashes }}

**capfirst** ---- {{ value|capfirst }}

**cut** -------- {{ value|cut:" " }}

**Namespacing**

It is something like providing the view name given in url.py file(not view name from models.spy file).

Generally used in dynamic url

e.g:-

<li><a href="{% url 'polls:detail' question.id %}">{{ question.question\_text }}</a></li>

**Forms in Django**

1. Bound form
2. Unbound form



**Advantages of Django forms:**

1. Validation of form data will be easy
2. We can develop form very easily
3. Creation of form, adding data to DB, validation of form data based on model/form is very easy.

Before learning more on Django form we will learn about csrf\_toekn

**csrf\_token**

The CSRF middleware and template tag provides easy-to-use protection against Cross Site Request Forgeries.

The best explanation we can get it from below link

<https://www.youtube.com/watch?v=hW2ONyxAySY>

let’s take an example how one can do it.

Suppose you are logged in to your bank account at same time you open a tab on your browser, that browser can trick with you back account(change password, fund transfer)

For creating any form in django create a class in forms.py file and use it in your view.py file for rendering this form.

from django import forms

class ForgotPassword(forms.Form):

    username=forms.CharField(label='Username')

    mobile\_number=forms.IntegerField(help\_text='Enter your registered mobile number', label='Mobile number')

class CapturePassword(forms.Form):

    password1=forms.CharField(min\_length=8,label="New Password", widget=forms.PasswordInput)

    password2=forms.CharField(min\_length=8,label="Confirm new Password", widget=forms.PasswordInput)

Some django form field options that we use frequently are:-

1. **required=Boolen -------->** to make filed as hidden field
2. **label="custom label" ------->** label name for field type.
3. **label\_suffix='some suffix for a label'**
4. **initial='some default or initial value'**
5. **help\_text='help text for your field'**

**widget**

<https://docs.djangoproject.com/en/3.0/ref/forms/widgets/>

A widget is Django’s representation of an HTML input element. The widget handles the rendering of the HTML, and the extraction of data from a GET/POST dictionary that corresponds to the widget

e.g—if want data of a input file to be hidden then use widget=forms.PasswordInput

**Widget types are:**

1. **TextInput**

Forms Widgets handling input of text

1. forms.NumberInput
2. forms.EmailInput
3. forms.URLInput
4. forms.PasswordInput
5. forms.Textarea
6. **Selector and checkbox widgets**
7. forms.CheckboxInput
8. forms.select

etc.

1. **File upload widgets**
2. FileInput
3. ClearableFileInput
4. Composite widgets
5. MultipleHiddenInput
6. SplitDateTimeWidget

#############################################

# Getting data from form #

#############################################

We can get data from any form using 'cleaned\_data' variable on form object.

Value=fm.cleaned\_data['field\_name'] ---------- This will give value associated with variable field\_name

**Note**:

cleaned\_data is predefined variable which contains value in dictionary format.

Variable name is something related to input field/label/field type/id name.

**Example:**

if request.method=="POST":

        print('in user\_login views name')

**fm=StudentForm(request.POST)**

**if fm.is\_valid():**

**print('forms is valid')**

**uname=fm.cleaned\_data['username']**

**passwd=fm.cleaned\_data['password']**

else:

#code for validation failed

**##############################################################**

**# Routing/Redirecting from one view function to other view function #**

**##############################################################**

We can route from one view to other view in many ways-

1. Using HttpResponseRedirect
2. Using redirect (django.shortcuts)
3. By calling view function with suitable parameters ----- return view\_function(request)

**2.) Using redirect (django.shortcuts)**

Call redirect function with request, url or view mapping from urls.py file.

def user\_logout(request):

    logout(request)

    return redirect('woodshophome:woodshophome-home')

return redirect(url\_or\_view mapping) --------- Synatx

return redirect('app\_name:alias\_of\_url'\_for\_specific\_view\_function) --------- Synatx

return redirect (reverse('appname:alias\_name', kwargs={"arg1": val1, "arg1": val2..}))

arg1, arg2 ---- these are parameter in urls (e.g --- removing hardcoded url)

**Note:**

1. When redirecting using redirect function with url name or url mapping of view in urls.py file then url changes.

Example:

Views.py file

def homepage(request):

    return render(request,'testapp/home.html')

    #return HttpResponse('this is home page')

def form\_page(request):

    if request.method=="POST":

        return redirect('testapp:home')

    else:

        return render(request,'testapp/myform.html')

urls.py file

app\_name='testapp'

urlpatterns = [

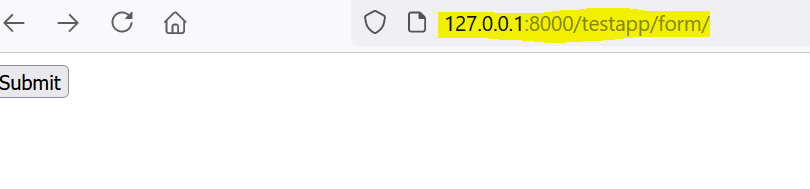
    path('home/', homepage, name='home'),

    path('form/',form\_page,name='form\_page')

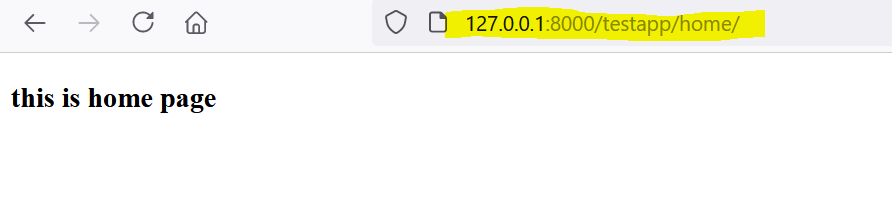
]

**Result:**

When accessing form\_page view function.



When clicked on submit form and homepage view is called.



**3.) By calling view function with suitable parameters**

Call the view function with request.

def homepage(request):

    return render(request,'testapp/home.html')

    #return HttpResponse('this is home page')

def form\_page(request):

    if request.method=="POST":

        return homepage(request)

    return render(request,'testapp/myform.html')

**NOTE:**

1. There will be no change in url when it will transfer control from form\_page view to homepage view.

################################

# GET vs POST # ------- asked in interview

################################

|  |  |
| --- | --- |
| GET | POST |
| > Get request is not secured because data is exposed in URL bar  > In GET request **data is sent in header**  > In case of Get request, only **limited amount of data can be sent** because data is sent in header.  >GET request is generally for fetching data from server. | > Post request is secured because data is not exposed in URL bar.  >In **POST data is sent in body**  > In case of post request**, large amount of data can be sent** because data is sent in body.  >POST request is generally for send data to server. |

**##############################**

**# Form validation #**

**##############################**

There are two types of validations.

1. Implicit validation (in-built) ---------- by Django validators, available in django.core.validators
2. Explicit validation ------ by programmers

Validation are always written in forms.py file for specific form class.

#################

#Implicit validators#

################

There are two types of implicit validators

1. Built in validators
2. Custom validators

#################

# Built in validators#

################

There are several validators available in django we need to just specify in form field type to use them.

e.g---

class EmployeeForm(forms.Form):

    ename=forms.CharField(**validators=[validator\_name])**

    esal=forms.FloatField()

#################

# Custom validators#

#################

We can define our own custom validators. Custom validators are nothing but any methods defined outside the model class and use that function for validators.

def my\_validator(value):

#some validation logic

return value

Class MyModel(forms..Form):

name=models.charfield(max\_length=100, validators=[my\_validator])

#

forms.py file

def salary\_validator(value):

    if value<10000:

        raise forms.ValidationError('salary should be more tan 10000')

    return value

class EmployeeForm(forms.Form):

    ename=forms.CharField()

    esal=forms.FloatField(validators=[salary\_validator])

    email=forms.EmailField()

    eid=forms.CharField()

views.py file

def emp(request):

    if request.method=="POST":

        form=EmployeeForm(request.POST)

        if form.is\_valid():

            return HttpResponse('form data is good')

        else:

            return render(request,'testapp/emp.html',{'form':form})

    if request.method=="GET":

        form=EmployeeForm()

        return render(request,'testapp/emp.html',{'form':form})

urls.py file

app\_name='testapp'

urlpatterns = [

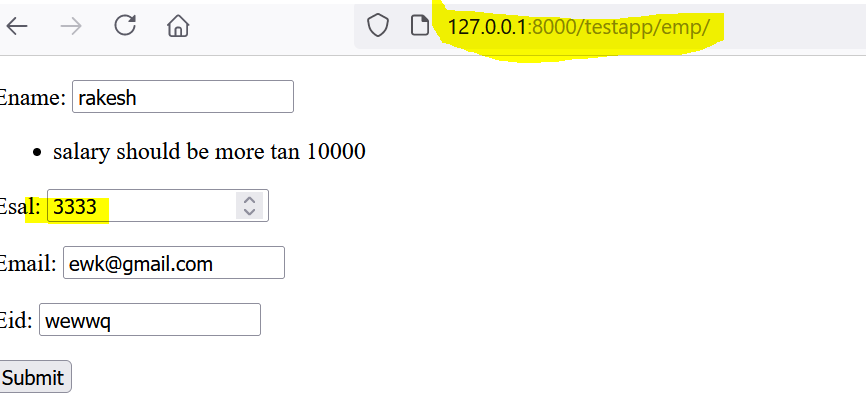
    path('home/', homepage, name='home'),

    path('form/',form\_page,name='form\_page'),

    path('emp/',emp,name='emp'),

]

Output: ( When we enter salary <10000 then we will get validation error message)



**#################**

**#Explicit validation#**

#################

We can do explicit validation in two ways –

1. Field validations
2. Object validation (Validating all field of form class)

Explicit validator are validator which are writer by programmer, not available by default.

We can define it in below way-

#####################

#Explicit field validation#

####################

Class MyForm(form.Form):

#form fields

def clean\_varname(self):

field\_values=self.cleaned\_data[‘field\_name’]

**#if validation fails**

**#raise forms.validationError(‘some message’)**

**# if validation logic success**

**Do something**

**return varname\_value**

**NOTE:**

1. clean\_varname ------ here varname is the name of variable/field of your form class that you want to validate
2. self.cleaned\_data[‘field\_name’] ---- this is way to get value of any field in clean\_varname().
3. In REST we have validate\_variablename(self,value) method for filed validations.

**Example:**

Write validation logic for ename field of below form class.

forms.py file

class EmployeeForm(forms.Form):

    ename=forms.CharField()

    esal=forms.FloatField()

    email=forms.EmailField()

    eid=forms.CharField()

Solutions:

Forms.py file (Only validating ename field)

class EmployeeForm(forms.Form):

**ename**=forms.CharField()

    esal=forms.FloatField()

    email=forms.EmailField()

    eid=forms.CharField()

    def clean\_**ename**(self):

        if len(self.cleaned\_data['ename'])>10:

            print('validation of employee name failed')

            raise forms.ValidationError("name can't be more than 10 character long")

        return self.cleaned\_data['ename']

**Note:**

1. Now whenever we call is\_valid() in views class it will internally run the validation for variable ename.
2. In function clean\_ename , I have used ename b/c I want to validate ename of form field.

Views.py file

def emp(request):

    if request.method=="POST":

        form=EmployeeForm(request.POST)

        if form.is\_valid():

            return HttpResponse('form data is good')

        else:

            return render(request,'testapp/emp.html',{'form':form})

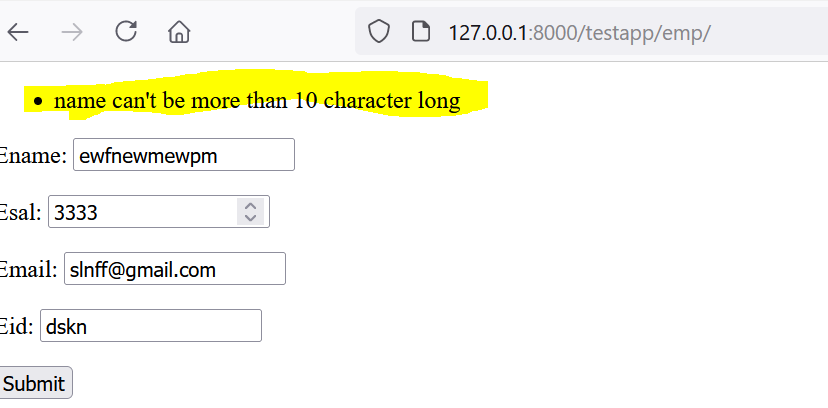
    if request.method=="GET":

        form=EmployeeForm()

        return render(request,'testapp/emp.html',{'form':form})

Result:

When we entered ename value more than 10 character long



Note:

We can use implicit validation and explicit validation both in a form class.

#######################

# Explicit object validation#

#######################

Explicit object validation (validation for all fields) are done using clean() function.

This function should be written in model class.

def clean(self):

clean\_data=super().clean() # data will be in dict format

field\_name1\_value=cleaned\_data[field\_name1]

field\_name2\_value=cleaned\_data[field\_name2]

#...

#validation logic for fields

field\_name --- these are name of form fields.

* In REST we have validate(self,value) method for object validations.

**Advantage of object validation:**

It saves effort for writing methods for each fields to do validation.

**Example:**

Write form validation logic in which length of name >0 and salary >10000.

**Solutions:** (fomrs.py file)

class EmployeeForm(forms.Form):

    ename=forms.CharField()

    esal=forms.FloatField()

    email=forms.EmailField()

    eid=forms.CharField()

    def clean(self):

        cleaned\_data=super().clean()

        ename=cleaned\_data['ename']

        esal=cleaned\_data['esal']

        if len(ename)==0 or esal<10000:

            raise forms.ValidationError('validation failed for name and salary')

############################

# Modal Form #

############################

Form created using any model is called model form. We can create is in below way-

Class FormClass(models.Model):

Class Meta:

model=model\_class\_name #model class name

fields= Fields of model form # denotes the field of modal form

We can define the fields of modal class in three ways-

1. fields='\_\_all\_\_'
2. fileds=[field\_name\_of\_modal\_class]
3. exclude=['name\_of\_field\_that\_don't\_want\_in\_form']

#####################

#Modal form Validation#

#####################

It is same as how we do in non-modal form validations. It’s same way as we do in REST.

**Question:**

Create a Employee modal form from below modal class and write validation logic for salary.

Modal.py file

class Employee(models.Model):

    ename=models.CharField(max\_length=100)

    esal=models.FloatField()

    email=models.EmailField()

    eid=models.CharField(max\_length=10)

forms.py file

class EmployeeForm(forms.ModelForm):

    class Meta:

        model=Employee

        fields='\_\_all\_\_'

    def clean\_esal(self):

        if self.cleaned\_data['esal']<1000:

            print('validation of salary fails')

            raise forms.ValidationError('salary can not be less than 1000')

        return self.cleaned\_data['esal']

views.py file

def update\_emp(request,id):

    emp=Employee.objects.get(id=id)

    if request.method=='POST':

        form=EmployeeForm(request.POST,instance=emp)

        if form.is\_valid():

            form.save()

            return HttpResponse('details updated')

        else:

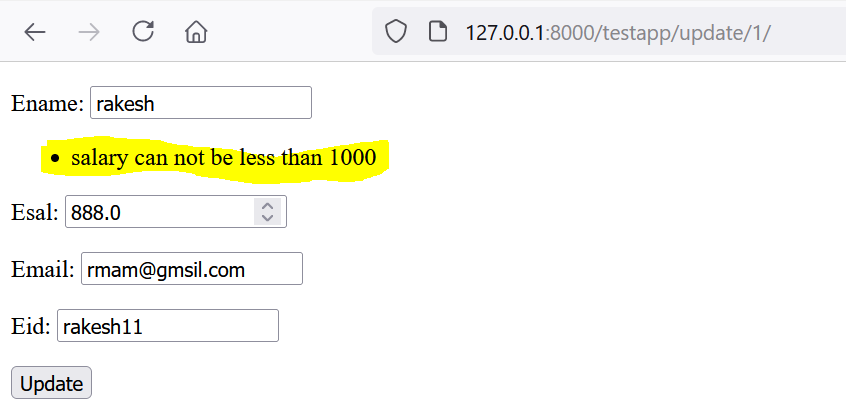
            return render(request,'testapp/update.html',{'form':form})

    else:

        form=EmployeeForm(instance=emp)

        return render(request,'testapp/update.html',{'form':form})

**Result:** (when supplying salary less then 1000 will get validation error)



############################################################################## Registering Model in admin file for admin panel #

#############################################################################

We can register any model/model class in admin file so that we can view/update/delete/add data for that model from admin panel. There are multiple ways to register it.

##############

# Method 1 #

##############

admin.site.register(Model\_name)

If we register in above fashion then we can’t see data for that model in list form.

##############

# Method 2 #

##############

class ModelNameAdmin(admin.ModelAdmin):

list\_display=[list\_to\_diaply/add/update/delete\_from\_admin\_panel]

admin.site.register(ModelName,ModelNameAdmin)

If we register model in above way then we can see data for that model in list form.

Example

class OrderAdmin(admin.ModelAdmin):

    list\_display=['order\_item','username','order\_status','order\_unit','order\_date']

admin.site.register(Order,OrderAdmin)

##################################

# Showing fields of model class in django#

##################################

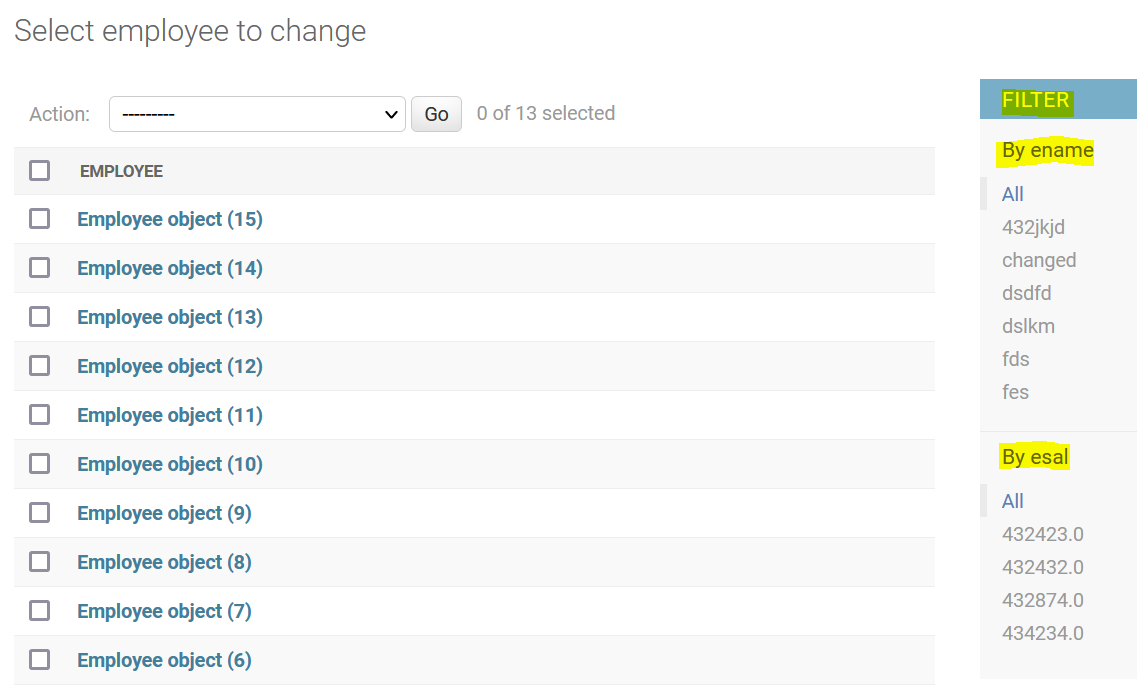
1. Using list\_dsplay ----- already done
2. Using list\_filter --- used for filtering data from lot of data/fields.

Once you've got a lot of items in a list, it can be useful to be able to filter which items are displayed. This is done by listing fields in the list\_filter attribute.

admin.spy

class EmployeeAdmin(admin.ModelAdmin):

    list\_filter=['ename','esal']



Now it’s allowing to filter data by ename and esal fields.

#############################

# Update operation in django form#

#############################

If we are doing update operation then we need to use 'instance' (specified the data/object whose data will be updated) keyword while creating the object.

form=StudentForm(request.POST,instance=stud\_instance) --- Syntax

obj.update(fd1=value1, fd2=value2 ..) --- django query for updating data, **alternate way**

**Note:**

If using this approach then form class should be a modal form.

**Questions:**

Update detail for employee modal class.

Modal.py file

class Employee(models.Model):

    ename=models.CharField(max\_length=100)

    esal=models.FloatField()

    email=models.EmailField()

    eid=models.CharField(max\_length=10)

forms.py file

class EmployeeForm(forms.ModelForm):

    class Meta:

        model=Employee

        fields='\_\_all\_\_'

views.py file

def update\_emp(request,id):

    emp=Employee.objects.get(id=id)

    if request.method=='POST':

        form=EmployeeForm(request.POST,instance=emp)

        if form.is\_valid():

            form.save()

            return HttpResponse('details updated')

    else:

        form=EmployeeForm()

        return render(request,'testapp/update.html',{'data':emp})

urls.py file

urlpatterns = [

    path('update/<int:id>/',update\_emp,name='emp'),

]

Update.html

<body>

    <form action="" method="POST">

        {% csrf\_token %}

        <input type="text" name="ename" value="{{ data.ename }}"><br>

        <input type="text" name="esal" value="{{ data.esal }}"><br>

        <input type="text" name="email" value="{{ data.email }}"><br>

        <input type="text" name="eid" value="{{ data.eid }}"><br>

        <input type="submit" value="Update">

    </form>

</body>

**Note:**

We can use instance keyword to show the data during the form creation.

form=MyForm(instance=data) ----- syntax

data must be either modal instance of queryset.

Modals.py ---- same as above

Forms.py file ---- same as above

Views.py file

def update\_emp(request,id):

    emp=Employee.objects.get(id=id)

    if request.method=='POST':

        form=EmployeeForm(request.POST,instance=emp)

        if form.is\_valid():

            form.save()

            return HttpResponse('details updated')

    else:

        form=EmployeeForm(instance=emp)

        return render(request,'testapp/showdata.html',{'form':form})

urls.py file

urlpatterns = [

    path('update/<int:id>/',update\_emp,name='emp'),

]

showdata.html

<body>

    {{form.as\_p}}

</body>

#######################

# Delete Operation of Data#

#######################

For deleting any data call delete method on object/instance which you want to delete.

entry=Entry.object.get(id=num)

entry.delete() --------- Syntax

**-:User permission in django:-**

In django we have built in template variable **'{ perms }'** which contains all the permissions for currently logged in user

When ever we create any model from models.py file , python automatically create 4 type of permission- add , delete , change, view.

This is an instance of django.contrib.auth.context\_processors.PermWrapper, which is a template-friendly proxy of permissions

**'{% if perms.app\_Name.permission\_modelnameOfModels.pyFile %}'**

----------------For checking permission for logged in user

----------------**Model name of models.py file will be changed to lower case**

**---------------- this hold good to use for templates**

**'{% if perms.app\_Name %}'**

To check is logged in user has any permission in app app\_Name

**user.get\_all\_permissions** ---- **to get all permission in template**s.

User.has\_perm(‘app\_name.model\_name’) ------ **to test permission in view function**

{% for each in user.get\_all\_permissions %}

{{ each }}

<br>

{% endfor%}

Assuming you have an application with an app foo and a model named Bar, to test for basic permissions you should use-

* **add: user.has\_perm('foo.add\_bar')**
* **change: user.has\_perm('foo.change\_bar')**
* **delete: user.has\_perm('foo.delete\_bar')**
* **view: user.has\_perm('foo.view\_bar')**

**----------above all are good to use in view function only not in template**

* **user.get\_all\_permissions()**-------- to get all permissions in views function, not in template

**Note:**

Model name of models.py file is changed to lower case while using in perms tag

**Example-**

Let say we have two user sonam and rahul. Rahul have add and view permission for model Blog and sonam have delet,view,add,change permission. WAP to diplay those permission on their profile page

**models.py file**

from django.db import models

# Blog model for permission learning

class Blog(models.Model):

    article\_name=models.CharField(max\_length=100)

    artile=models.CharField(max\_length=200)

**Template file**

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>User Profile</title>

</head>

<body>

    <h3>Welcome to profile page,{{ name }}</h3>

    <a href="{% url 'user\_logout' %}">Louout</a>&nbsp;

    <a href="{% url 'change\_password' %}">Chnage Password</a>

    <br>

    {% if perms.login.view\_blog %}

        <p>You have view permissions</p>

        <br>

    {%  endif %}

</body>

</html>

**-: Session Management using COOKIES in django:-**

A cookie is a small piece of information which is stored in the client browser. It is used to store user's data in a file permanently (or for the specified time).

Cooke values are always string type.

**HttpRequest.COOKIES** ---- gives all cookies in key, value form(dictionary)

**HttpRequest.COOKIES[key]** ----------- gives cookies value for cookie name key

**Creating cookies**

**response.set\_cookie(key,vallue='',max\_age=None,expires=None,path='/',domain=None,secure=False,htttponly=False,samesite=None)**

----- This is the methods used for creation cookies, except key all are non-mandatory arguments

response --- this is response being returned to client, necessarily it will be HttpResponse (Check in below e.g)

max\_age---- age of cookies in seconds. Default=None and age is till browser is open

expires---describes the time when cookies expires. It could be datetime object

httponly--- use this only when don’t want server side javascript to have access to cookies.

*If cookie data is more than 4096 bytes then cookie will not work and no error or exception will be raised*

**Note:**

For creating any session/cookie and setting/updating data.

1. Fist get the data that you want to return and store in one variable ( say --- response)
2. Update session/cookie for on that response.
3. Finally return the output stored in variable.

------ You can these steps in below example

**Reading/Accessing the cookie data**

**request.COOKIES[key\_name]**---- reading data from cokies for given key\_name

**request.COOKIES.get(key\_name,default)**----if key is present in cookies then give data for that cookie else give default value

**Replace/Append cookies**

When we change/assign the new value to a given cookie then value if replaced

**Question:**

WAP to display count or number of times a user visits the page.

**Solution**:

Views.py file

def get\_count(request):

    if 'count' in request.COOKIES:

        newcount=int(request.COOKIES['count'])+1

    else:

        newcount=1

    response=render(request,'testapp/count.html',{'count':newcount})

    response.set\_cookie('count',newcount) #increase the count of session

    return response

urls.py file

urlpatterns = [

    path('visit/',get\_count,name='get-visit-count'),

]

count.html

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h3>Page visit count is: {{count}}</h3>

</body>

</html>

**Deleting the cookie**

**HttpResponse.delete\_cookie(key,path='/',domnain=None)**---- this is used for deleting the cookie

**Creation Signed Cookies**

HttpResponse.set\_signed\_cookie(key,value,salt='',max\_age,expirese=None,path='/',domain=None,secure=False,httponly=False,samesite=None)



----this si used for creating the signed cookies. Becareful about the salt value, it should be same while accessing the cookies

**Reading Signed Cookies**

HttpResponse.get\_signed\_cookie(key,salt='',max\_age=None,default=RAISE\_ERROR)

Return the value for given key or raise django.core.singing.BadSignature exception if singature is no longer valid

**Drawback of Cookies:**

1. By using cookies, we can send very less amount of information (max 4 KB).
2. Cookies have security concern.
3. Cookie info is always string type.
4. On every request browser have to send all cookie information, that may cause network traffic.
5. Maximum number of cookies supported by browser is always fixed.

**-:SESSION in django:-**

A session is a mechanism to store information on the server side during the interaction with the web application.

The session is a semi-permanent (depending on age of session) and two-way communication between the server and the browser

The two-way communication means that every time the browser/client makes a request, the server receives the request and cookies containing specific parameters and a unique Session ID which the server generates to identify the user. The **Session ID doesn’t change for a particular session**, **but the website generates it every time a new session starts**

*By default session name is sessionid*

**Types of sessions**

Based on storage mechanism of session it can be classified broadly in 4 types:-

1. database-backed sessions---- by default, 'django.contrib.sessions’ add in setting.py file
2. cached-backed sessions -----
3. file-backed session ---- django.contrib.sessions.backends.file add in installed apps in setting.py file
4. cookie-based sessions-- django.contrib.sessions.backends.signed\_cookies add in installed apps, that it
5. **Database-backed sessions**

If you want to use a database-backed session, you need to add 'django.contrib.sessions' to your INSTALLED\_APPS setting

**Note:**

* If using database-backed session then we need to run migrate command so that session table is created.
* Session is stored on server side so there are no method in session that needs response object.

**Creating/reading/modifying the sessions**

request.session ------- Dictionary containing all session

**request.session[‘key’]=value**-----creating session for given key and value

**request.session.get(‘key’,default=’val’)**---returns the value for given key from session

**request.session.get(key,default=default\_val)**--- if given key exist in session return value for that key else default

**del request.session[‘key’]**----deleting the session

**‘key’ in request.sessions**-------------checking the existence of sessions

**NOTE:**

*Session objects are created or retrieved from request object only*.

request.session -- returns session as a dictionary on this we can apply dictionary method to get required data

* **keys()**-- returns a view object that contains a list of all keys in the dictionary
* **items(**)--same as dictionary but here for session object
* **clear()** --same as dictionary but here for session object
* **flush()**---Deletes the current session data from the session and deletes the session cookie

*All above methods are applicable on session dictionary. i.e request.session.keys()*

***Some useful methods of sessions***

**request.get\_session\_cookie\_age()---** gives the age of session cookies. default to **SESSION\_COOKIE\_AGE(default 2 weeks)** from setting file.

**request.set\_expiry(value)** --- sets the session expiry.

* value=integer, it presents the session life that number of seconds
* value=datetime object, then it expire at that time
* value=0, then it expires when browser is closed
* value=None, then use globaly expiry policy

**request.get\_expiry\_date()** ---returns the date the session will expires

**request.clear\_expired()**--- removes the expired session from session stores

**request.get\_expiry\_age()**--- returns the number of seconds until the session expires

**request.set\_test\_cookie()**--- Sets a te3st cokkie to determine if user's browser supports cookie,Due to the way cookies work, you won’t be able to test this until the user’s next page request

**request.test\_cookie\_worked()**--- Returns True if user's browser accepted test cookie

**request .delete\_test\_cookie()** --- to delete the test cookie.

**Example:**

WAP to get the count of number of times any user visits website.

**Solution**

def page\_count\_session(request):

    if 'count' in request.session:

        newcount=request.session['count']+1

    else:

        newcount=1

    request.session['count']=newcount

    return render(request,'testapp/count.html',{'count':newcount})

**File Based session**

To use file based session add below lines in setting.py file, nothing else to touch

SESSION\_ENGINE='django.contrib.sessions.backends.file'

SESSION\_FILE\_PATH=full\_path\_for\_session\_file

**Some useful setting in setting.py file for session**

Below are some parameters that we can use on requirement basis.

1. **SESSION\_COOKIE-AGE**--- used to set the default age of cookie
2. **SESSION-COOKIE\_DOMAIN**--- the domain to use for session cookies
3. **SESSION\_CACHE\_ALIAS**---used if using the cached based sesion storage, dfault is False, this selects the cache to use
4. **SESSION\_COOKIE\_HTTPONLY**--Whether to use HttpOnlyfalg on the session cookie. Default- false
5. **SESSION\_COOKIE\_NAME='str'** -- name that you want fpr session, default is sessionid
6. **SESSION\_COOKIE\_PATH=’str’**-- The path set on the session cookie
7. **SESSION\_FILE\_PATH='str'** -- path for session file is using file based sessions
8. **SESSION\_EXPIRE\_AT\_BROWSER\_CLOSE= 'bool'** -- Whether to expire the session when user closes the browser

**Creating session and cookies**

HttpResponse.set\_cookies[key,value,max\_age..] --Creating cookies

request.session['key']=value --- creating session

**Getting cookie and session**

HttpRequest.cookies----- get all cookies in python dictionary format

request.session --- get all session data

request.session.get['key','value'] ----- to get session data

**Deleting cookie and session**

HttpResponse.delete\_cookie(key,path='/',domnain=None)

del request.session['key'] ---- delete session data for given key

del request.session ---- to delete all session data

**-:CACHE:-**

Cache is something like keeping it ready or storing temporarily for any request to make website faster.

It could be image, web page or any other file.

Django supports three types of caching.

1. **Database caching**

CACHE={

'default':{

'BACKEND':'django.core.cache.backends.db.DatabaseCache',

'LOCATION':'my\_cache\_table',

}

}

1. **File system caching**

CACHE={

'default':{

'BACKEND':'django.core.cache.backends.filebased.FileBasedCache'

'LOCATION':'c:/django....'

}

}

1. **Local Memory caching**

CACHES={

'default':{

'BACKEND':'django.core.cache.backends.locmem.LocMemCache',

'LOCATION':'unique-snowflake',

}

}

**How to implement caching**

We can implement caching in three ways

1. **the per-site caching** ---- Caching entire website
2. **The per-view cache** ---Caching required view
3. **Template fragment caching** ---caching templates. It gives more control what to cache

**The per-site cache**

Once the cache is set up the simplest way to use caching is to cache is entire site.

Make the below entry in you setting.py file in same order as mentioned below.

Below entries are just to instruct python to work for per-site caching

MISSLEWARE=[

'django.middleware.cache.UpdateCacheMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.cache.FetchFromCacheMiddleware',

]

**CACHE\_MIDDLWWARE\_ALIAS** --- the cache alias to use for storage

**CACHE\_MIDDLEWARE\_SECONDS** --the number of seconds each page should be cached.

**CACHE\_MIDDLEWARE\_KEY\_PREFIX** --- used to prevent the key collisions, in case cache is share with multiple website

**database caching for per-site**

django store the cached data in your DB. This works best if you have fast,well indexed DB server.

Make the below entries in your setting.py file

CACHE={

'default':{

'BACKEND':'django.core.cache.backends.db.DatabaseCache',

'LOCATION':'my\_cache\_table',

}

}

'my\_cache\_table’ --- this is name of cache table. You can give any which you want.

*This is just to instruct that I want database caching, for per site, per view we need to make other entries.*

MISSLEWARE=[

'django.middleware.cache.UpdateCacheMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.cache.FetchFromCacheMiddleware',

] **# Add these entries in middleware section to instruct that we want entire site caching**

Before using the database cache you must create the DB cache table by below commands.

**python manage.py createcachetable**

---------- This creates the DB table from the location value

----------- If using multiple cache table then it creates for each cache

**Cache Arguments**

TIMEOUT--- the default time out to use for cache. default 300 seconds.0 means no caching

MAX\_ENTRIES

CULL\_FREQUENCY

CULL\_FREQUENCY

CACHE={

'default':{

'BACKEND':'django.core.cache.backends.db.DatabaseCache',

'LOCATION':'my\_cache\_table',

‘TIMEPOUT’:60

‘OPTION’:{

‘MAX\_ENTRIES’:1000

}

}

}

**FileSystem caching for per-site**

The file based backend serializes and stores each cache value as a separate file

Add below entries in yousetting.py file

CACHE={

'default':{

'BACKEND':'django.core.cache.backends.filebased.FileBasedCache'

'LOCATION':'c:/django....'

}

}

'LOCATION':'c:/django....' ---- this is absolute path of file where cached will be stored. Create it before using

Remaining all code and concepts are same as BD-backed caching.

**Local memory caching**

This is default caching it no caching is specified

Don't use in production environment

Here cached data is stored in local computer memory so can't be seen but can be verified

Make the below entries in the setting.py file.

CACHES={

'default':{

'BACKEND':'django.core.cache.backends.locmem.LocMemCache',

'LOCATION':'unique-snowflake',

}

}

**The per-view caching**

Caching the individual views.

django.views.decorators.cache defines **cache\_page decorator** that will automatically cache the view's response.

If multiple URLs point at same view function, each URL will be cached separately (e.g—dynamic URLs)

*We need to make below entries as required place (urls.py file and views.py file) for using per view caching apart from this we need to also specify where cache data will be stored*.

**Using per view from view name**

Import cache\_page and use it with view function name

from django.view.decorators.cache import cache\_page

@chache\_page(timeout,cache,key\_prefix)

def my\_views(request):

#

**specifying per-view cache in URL conf**

import cache\_page and use it with url pattern

from django.views.decorators.cache import cache\_page

urlpatterns=[

path('url\_pattern',cache\_page(timeout,cache,key\_prefix)(view\_finction),name=xyz

]

*timeout, cache, key\_prefix are same as previous*

**Database caching for per-view**

It will be same as what we have used in the per-site caching with respect to required library conceptually

django store the cached data in your DB. This works best if you have fast,well indexed DB server.

Make the below entries in your setting.py file

CACHE={

'default':{

'BACKEND':'django.core.cache.backends.db.DatabaseCache',

'LOCATION':'my\_cache\_table',

}

}

'my\_cache\_table’ --- this is name of cache table. You can give any which you want

Before using the database cache you must create the DB cache table by below commands.

**python manage.py createcachetable**

---------- This creates the DB table from the location value

----------- If using multiple cache table then it creates for each cache

**Cache Arguments**

TIMEOUT--- the default time out to use for cache. default 300 seconds.0 means no caching

MAX\_ENTRIES

CULL\_FREQUENCY

CULL\_FREQUENCY

CACHE={

'default':{

'BACKEND':'django.core.cache.backends.db.DatabaseCache',

'LOCATION':'my\_cache\_table',

‘TIMEPOUT’:60

‘OPTION’:{

‘MAX\_ENTRIES’:1000

}

}

}

Views.py file

@cache\_page(timeout=20)

def signup(request):

        return render(request,'login/signup.html',form)

singup.html

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <form action="" method="POST">

        {% csrf\_token %}

        <!--{{ form.as\_p }}-->

        {% for fm in form %}

        {{ fm.label\_tag}}{{ fm }}

        <br>

        {% endfor %}

        <input type="submit" value="submit">

    </form>

    <a href="{% url 'login-user\_login' %}">Login</a>

</body>

</body>

</html>

*Explanations:*

Here this will cache two page 1.)Signup 2.)Login page b/c we have created link for login in signup page and that should happen as per per-view caching definitions]

Using URLconf

urlpatterns = [

    path('', cache\_page(60)(views.signup), name='login-signup\_home'),

except this remaining all will be same, this will al

**FileSystem caching for per-view**

The file based backend serializes and stores each cache value as a separate file

Add below entries in yousetting.py file apart from it we need to also specify the per view caching which is already done at start of per view caching.

CACHE={

'default':{

'BACKEND':'django.core.cache.backends.filebased.FileBasedCache'

'LOCATION':'c:/django....'

}

}

'LOCATION':'c:/django....' ---- this is absolute path of file where cached will be stored. Create it before using

Remaining all code and concepts are same as BD-backed caching

**Local memory caching**

This is default caching it no caching is specified

Don't use in production environment

Here cached data is stored in local computer memory so can't be seen but can be verified

Make the below entries in the setting.py file.

CACHES={

'default':{

'BACKEND':'django.core.cache.backends.locmem.LocMemCache',

'LOCATION':'unique-snowflake',

}

}

**FileSystem caching for per-site**

The file based backend serializes and stores each cache value as a separate file

Add below entries in yousetting.py file

CACHE={

'default':{

'BACKEND':'django.core.cache.backends.filebased.FileBasedCache'

'LOCATION':'c:/django....'

}

}

**Template Fragmentation Caching**

I have to make notes of it

**-:LOW LEVEL CACHE API:-**

Sometimes, caching an entire rendered page doesn’t gain you very much and is, in fact, inconvenient overkill. Perhaps, for instance, your site includes a view whose results depend on several expensive queries, the results of which change at different intervals. In this case, it would not be ideal to use the full-page caching that the per-site or per-view cache strategies offer, because you wouldn’t want to cache the entire result (since some of the data changes often), but you’d still want to cache the results that rarely change.

For cases like this, Django exposes a simple, low-level cache API. You can use this API to store objects in the cache with any level of granularity you like. You can cache any Python object that can be pickled safely: strings, dictionaries, lists of model objects, and so forth. (Most common Python object

**Setup/types of low level caching**

1. Database caching
2. File system caching
3. Local memory caching

-------- to set these setup we need to follow the same step what we have learnt.(exactly)

**How to set cache:**

Import cache from django.core.core.cache and use set method to cache.

from django.core.core.cache import cache

cache.set(key,value, timeout=DEFAULT\_TIMEOUT, version=None)

key ---- it shod be str

value - it can be any pickled python objetc

timeout -- number of second to timout the the cache

version---- it is an int. You can set the cache with same key but different version

**How to get cache**

cache.get(key,default=None, version=None)

---- This is used to get the cache. If key not present then return the default value

---- If key doesn't exists it returns None

**Other useful methods**

* cache.get\_or\_set(key, default, timeout=DEFAULT\_TIMEOUT)– if key was present return that else set for given key are return that value
* cache.add(key,value,timeout=DEFAULT\_TIMEOUT)
* cache.set(dict,timeout)--set many cache at a time
* cache.get\_many(keys,timeout) --- to get many cache at a time with gven keys
* cache.delete(key,version+none) --- key keys explicity to clear cache
* cache.delete\_many(keys,defailt=None)
* cache.cear() --- clear all keys/data from cache
* cache.touch(key, timeout=DEFAULT\_TIMEOUT) --- used to set timeout for key, retuns true it operation was succ.
* cache.iner(key,delta=n,version=None) ---- increase value for given key
* cache.decr(key, delta=n, version=None) --- decrease value for given key

**##################################################################################################**

**#**  **-:SIGNALS:- #**

#############################################################################

The signals are utilities that allow us to associate events with actions. We can develop a function that will run when a signal calls it

In many cases when there is a modification in a model’s instance, we need execute some action. Django provides us an elegant way to handle with these situations with help of django signals

Most common singals are:

**pre\_save/post\_save**: This signal is thrown before/after the method save()

**pre\_delete/post\_delete**: Before after delete a model’s instance (method delete()) this signal is thrown

**pre\_init/post\_init**: This signal works before/after instantiating a model (\_\_init\_\_() method)

**Types of signals:**

There are two types of singals in django

1. **built-in signals**--- login, logout, pre\_save, post\_save etc
2. **custom signals** ---

**Note:**

For creating any signals or working with any signals, create a signals file in your app in order to make code cleaner

**How to connect signals**

1. **@receiver decorator**

Import receiver from django.dispatch and use it with signals definition

1. **@receiver(signal\_name, sender=MyModel)***--- in case you want to connect for specific model/user/sender*
2. **@receiver(signal\_name)**---- connecting for any sender

@receiver can be one or multiple signals based on demands

*@receiver(post\_save, sender=Model1)*

*@receiver(post\_save, sender=Model2)*

*@receiver(post\_save, sender=Model3)*

*def my\_signal\_handle(sender , \*\*kwargs)*

*# some code here*

**Signal handler function***:*

**def signal\_function\_handle(sender,instance,user,\*\*kwargs):**

**#**

**def my\_callback(sender, \*\*kwargs):**

**#**

@receiver(post\_save)

def update\_complete(sender,\*\*kwargs):

    print('data saved succesfuly')

1. **Using connect()**

Use the signals name in conjuction with connect(function\_name\_for\_signal,sender)

**signal\_name.connect(function\_to\_handle\_signal, sender=MyModel)**

**request\_finished.connect(my\_reciver\_function)**

Overwrite the ready method in apps.py file in your appconfig class

**def ready(self):**

**import app\_name.signals**

default\_app\_config=’application\_name.apps.appnameconfig ------ In \_\_init\_\_() file

**appnameconfig ---** this is predefined class created while creating application

**Note:**

Above line should be written outside the function\_to\_handle\_signla

e.g-

post\_save.connect(my\_function\_post\_save, sender=MyModel)

**Defining signal\_handler or receiver function**

Define your handler function having **sender(mandatory)**,request,user and **\*\*kwargs(mandatory)** as argument.

Arguments of receiver function depends on the signals

**def signal\_function\_handle(sender,instance,user,\*\*kwargs):**

**#**

**def my\_callback(sender, \*\*kwargs):**

**#**

sender----- signal sender

instance ---- It represents object on which we did something

request--- HttpRequest object

user--- django user object

\*\*kwargs--- keyword arguments

Def my\_function\_post\_save(sender,\*\*kwargs):

# do the action…

**CUSTOM SIGNALS**

**Steps:**

Create instance of Signal class or create a signal, import from django.dispatch

Create the signal receiver

Bind the signal

Send the signal

**NOTE:**

If using custom signal then we will have to create signal and send/generate signal manually.

We should always create signal in a sperate file to make it cleaner.

**Creating signal**

signal\_name= Signal(providing\_args=list)

the providing\_args is a list of names of arguments the signal will provide to listener

this is purely documentational, however as there is nothing that checks that signal actually provide these arguments to listener

**Sending signal**

For sending signal we can use send(sender, \*\*kwargs) or send\_robust(sender, \*\*kwargs) methods

send() propagates any exception if occurs in the receiver function

By default built-in signal uses send().

We send singal manually on some specific condition, e.g—send signal when form validation completes (fm.is\_valid() )

**Note:**

1. We should define the handler method and singal in same file (signals.py)
2. signal\_name.send(sender=’XYZ’) ---- this is how we send signal to handler method.

Example

Signals.py file in app

#creating notification signal

from django.dispatch import Signal, receiver

notofication=Signal(providing\_args=['rakesh','my\_nm']) #creating singal, notification

@receiver(notofication) # hadling notification signal

def show\_notification(sender=None, \*\*kwargs):

    print('show notification is called')

    print(sender)

    print(kwargs)

views.py file

def user\_login(request):

    signals.notofication.send(sender='RaKESH') #sending signal

Whenever the user\_login view wil be called it will run the notification signal

#############################################################################

#  **-:MIDDLEWARE:-** #

#############################################################################

Middleware is a lightweight plugin that processes during request and response execution. Middleware is used to perform a function in the application. The functions can be a security, session, csrf protection, authentication etc.

Whenever request come each time middleware is executed (based on type of middleware).

**Function based middleware**

Follow below step or procedure

def my\_middleware(**get\_response**):

#One-time config and initilization code

def my\_function(request):

#code to execute for each request/response before view called

response=get\_response(request)

#code to be executed for each request/response after view is called

return response

return my\_function

**Function based decorator in python**

def my\_decorator(func):

    #some codes

    def wrapper(\*args,\*\*kwargs):

        #some codes or actions

        return func(\*args,\*\*kwargs)

    return wrapper

The **get\_response** is callable provide by django, it might be actual view or the next middleware in the middleware chain

Note:

*Code after line response=get\_response(request) is called only when different view is called than previous call*

**Activating the middleware**

To activate middleware component, add it to the MIDDLEWARE list in setting.py file of project, with full path of middleware/middleware function

The order of middleware matter i.e. order of middleware declaration b/c middleware may depends on another middleware

Example:-

Middleware.py file in app

def my\_middleware(get\_response):

    print('One-time config and initilization code')#executed only once

    def my\_function(request):

        print('request/response before view called')#before each view function

        response=get\_response(request)

        print('afterrequest/response after view is called')#after each different view func.

        return response

    return my\_function

setting.py file

MIDDLEWARE = [

    'django.middleware.security.SecurityMiddleware',

    'django.contrib.sessions.middleware.SessionMiddleware',

    'django.middleware.cache.UpdateCacheMiddleware',

    'django.middleware.common.CommonMiddleware',

    'django.middleware.cache.FetchFromCacheMiddleware',

    'django.middleware.csrf.CsrfViewMiddleware',

    'django.contrib.auth.middleware.AuthenticationMiddleware',

    'django.contrib.messages.middleware.MessageMiddleware',

    'django.middleware.clickjacking.XFrameOptionsMiddleware',

    'login.middlewares.my\_middleware'

]

Views.py file

def signup(request):

    print('signup views')

def signup(request):

    print('signup views')

**Class Based middleware**

Creation of class based middleware is same as function based.

A simple genral layout is written below

class MyMiddleware:

def \_\_init\_\_(self,get\_response):

self.get\_response=get\_response

#One-time configuration and initialization

def \_\_call\_\_(self,request):

#Code to be executed for each request before the view are called

response=self.get\_response(request)

#code to be executed for each request/response after the view is called

return response

class Based Decorator:

class MyDecorator:

    def \_\_init\_\_(self,func):

        self.func=func

    def \_\_call\_\_(self, \*args,\*\*kwds):

        return self.func(\*args,\*\*kwargs)



**Activating the middleware**

To activate middleware component, add it to the MIDDLEWARE list in setting.py file of project, with full path of middleware/middleware function

The order of middleware matter i.e. order of middleware declaration b/c middleware may depends on another middleware

Note:

*response=self.get\_response(request)*calls the next middleware in case multiple middleware are installed/created in your project.

In case if you don’t want next installed middleware to execute then assign HttpResponse object to response variable. i.e --- response=HttpResponse(‘some text/url’)

**Middleware Hooks**

Only for class based middleware

1. process\_view(request,view\_func,view\_args,view\_kwargs)



1. process\_exception(request,exception)



1. process\_template\_response(request,response)



###################################################################################

# **Decorators in Django** #

###################################################################################

Django come with some built-in decorators, like login\_required, require\_POST or has\_permission

We can use them in views function.

@decorator\_name

def my\_view(request):

#

1. **login\_requried/@login\_requried**

This decorator is used for asking user to login before going to any specific url or page if he hits directly in browser, this decorator is present in **django.contrib.auth.decorators**

This makes user to login and then transfer to the page where her wanted to go by hitting directly in url

e.g—let say user hits profile page and he is not logged in then we can use @login\_required decorator on top of profile views function.

This decorator is used with **'LOGIN\_URL**' in static file. It’s value is url of login page(removing the server IP:port\_number)

e.g.

setting.py file

LOGIN\_URL='/woodshophome/login'

urls.py file

app\_name='woodshophome'

urlpatterns = [

    path('login/', views.user\_login, name='login'),

]

**How to use it:**

* Set value of ‘LOGIN\_URL’ in setting.py file
* In login view check if there is any value associated with ‘next’ key for GET request if yes then transfer to that page else transfer where you want

**Question**

We have one shopping website in which ask user to login in case he tries to fetch order details without login and once he login then send user to order details page

views.py file

from django.contrib.auth.decorators import login\_required

def user\_login(request):

    if request.method=="POST":

        form\_data=request.POST

        fm=AuthenticationForm(request=request,data=request.POST)

        if fm.is\_valid():

            username=fm.cleaned\_data['username']

            password=fm.cleaned\_data['password']

            user = authenticate(username=username, password=password)

            if user is not None:

                login(request, user)

                print('user id and password is correct')

**if request.GET.get('next',None): #check if there is next in url**

**return HttpResponseRedirect(request.GET['next'])**

**else:**

**return render(request,'woodshophome/home1.html')**

        else:

            print('invalid form, rendering login form again')

            form={'form':fm,}

            return render(request,'woodshophome/login.html',form)

    else:

        print('rendering login form')

        fm=AuthenticationForm(request=request,data=request.POST)

        form={'form':fm}

        return render(request,'woodshophome/login.html',form)

@login\_required

def order\_status(request):

#code for what you want to do

if request.GET.get('next',None) ---- checks if there is GET methods called for login page

return HttpResponseRedirect(request.GET['next']) --- this will transfer to page where user wanted to go

**-:Class Based View:-**

Class-based views provide an alternative way to implement views as Python objects instead of functions.

**They do not replace function-based views**.

**It’s introduced in django 1.3**

Simple and common operation we can do using CBVs easily but for complex operation we should go for function based view.

Function based view are more powerful compared to CBVs.

**Advantage**

1. Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching
2. Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

**Types of class-based view**

1. Base Class-Based views/Base view
2. Generic class-based views/Generic view

<https://www.youtube.com/watch?v=1dDimoXKb-g>

**Base Class Based Views/Base View**

It can be thought of as parent views, which can be used by themselves or inherited from.

They provide all capabilities required for projects, in which case there are Mixins which extend what base can do. It provides below three classes

1. **View** ---- this is used as parent class for creating class for class based view
2. **RedirectView** ---Redirects to a given URL
3. **TemplateView** ---Renders a given template, with the context containing parameters captured in the URL

**Method Flow chart for above three class**

All above three classes follow same method flow chart as below. Below methods are available in all three classes

1. setup()
2. dispatch()
3. http\_method\_not\_allowed()
4. get\_context\_data()

**as\_view(agr1, agr2,…) method**

While creating class based view we will be using this method for creating URL mapping

It returns a callable view that takes a request and returns a response

response = MyView.as\_view()(request)

**-:Creating class Based view:-**

We can use below steps for creating class based view.

**Steps**

1. Create your own view class(say MyView) and inherit built-in 'View'(from django.views) class
2. Write your required view class methods(get(), post(), put()) and your own business logic

**Http Methods supported by class based view**

#Http Methods# #CBV methods#

1. GET --------------- get(request,\*args,\*\*kwargs) ---- \*args and \*\*kwargs are optional
2. POST --------------- post(request,\*args,\*\*kwargs) ---- \*args and \*\*kwargs are optional
3. PUT --------------- put(request,\*args,\*\*kwargs) ---- \*args and \*\*kwargs are optional
4. trace()
5. delete()

etc

**Passing arguments to View Class**

We can pass arguments/value to View class using parameters/value in as\_view() but all those arguments must be available/declared in the class if no then error.

**CASE1:**

Passing arguments which is declared in view class

class MyView(View):

    name='Rakesh'

    def get(self,request):

        return HttpResponse('this is class based view')

urlpatterns = [

    path('class/',views.MyView.as\_view(name='Rakesh'), name='classview'),

    url(r'^$',views.handler,name='pagenotfound'),

]

No error while passing value of argument 'name' variable as it is declared in view class

**CASE2:**

Passing arguments which is not declared in view class

class MyView(View):

    name='Rakesh'

    def get(self,request):

        return HttpResponse('this is class based view')

urlpatterns = [

    path('class/',views.MyView.as\_view(age='Rakesh'), name='classview'),

    url(r'^$',views.handler,name='pagenotfound'),

]

Here it will throw error as we are passing age parameter/argument which is not declared in view class

When the view is called during the request/response cycle, the **setup()** method assigns the HttpRequest to the view’s request attribute, and any positional and/or keyword arguments captured from the URL pattern to the args and kwargs attributes, respectively. Then **dispatch()** is called

**dispatch(request, \*args, \*\*kwargs)**

The view part of the view – the method that accepts a request argument plus arguments, and returns a HTTP response

The default implementation will inspect the HTTP method and attempt to delegate to a method that matches the HTTP method; a GET will be delegated to get(), a POST to post(), and so on.

Example:

Let’s take as example of class based view

Views.py file

class MyView(View):

    name='Rakesh'

    def get(self,request):

        return HttpResponse('this is class based view')

urls.py file

urlpatterns = [

    path('class/',views.MyView.as\_view(name='Rakesh'), name='classview'),

    url(r'^$',views.handler,name='pagenotfound'),

]

**Passing context in class based view and rendering template**

We can pass it as we do in function based view, take a look at below example

class MyView(View):

    name='Rakesh'

    def get(self,request):

        context={'msg':'rendering template from class'}

        return render(request,'first/clsbsdtemplate.html',context)

        #return HttpResponse('this is class based view')

Nothing else need to change anywhere, remaining all code will be same

**Contact form using class based view**

Forms.py file

from django.views import View

class ContactForm(forms.Form):

    name=forms.CharField(max\_length=200,label='Enter your name')

    mobile=forms.IntegerField(label='mobile number')

Views.py file

class ContactFormClassBasedView(View):

    def get(self,request):

        fm=ContactForm()

        form={'form':fm}

        return render(request,'first/contactform.html',form)

    def post(self,request):

        fm=ContactForm(request.POST)

        if fm.is\_valid():

            print(fm.cleaned\_data['name'])

            return HttpResponse('form submitted')

Write temple file by own.

Note:

If we are using/sending parameter in CBV that parameter is not received during class creation but it’s received on respective methods.

Example:

class EmployeeView(View,id): # this is incorrect, class will not accept any parameter

    def get(self,request):

        return HttpResponse('this is get function of EmploueeView')

class EmployeeView(View):

    def get(self,request,id): #this is correct as methods will accept arguments/parameter

        return HttpResponse('this is get function of EmploueeView')

**-:Class Based view Using TemplateView:-**

Template view is available in 'django.views.generic.base' module

In template view no need to render any template django will itself take care

In template view 'template\_name' is built in variable. This must be assigned with a template name wich you want to render.

Views.py file

class MyTemplate(TemplateView):

    name='Rakesh'

    template\_name='first/clsbsdtemplate.html'

*template\_name ---- this is built-in variable. this is name of template that you want to render, don’t use any other variable and no need to render the template*

urls.py file

path('temp/',views.MyTemplate.as\_view(), name='template-view'),

**Note:**

We can render template directly from urls.py file (using template\_name vraible in as\_view() ), without hitting to views.py file.

urlpatterns = [

    path('temp1/',views.MyTemplate.as\_view(template\_name='first/clsbsdtemplate.html'), name='template-view'),

    url(r'^$',views.handler,name='pagenotfound'),

]

**Passing context to TemplateView**

We can pass any number of context variable and context data in below way.

For passing any context to template view we need to implement get\_context\_data(self, \*args, \*\*kwargs) method

How to implement:

def get\_context\_data(self, \*args, \*\*kwargs):

context = super(TemplateViewClass\_name,self).get\_context\_data(\*args, \*\*kwargs)

context[var1] = 'value'

context['var2'] = self.variable #this is for class vaiable

return context

**Example 1**

Views.py file

class MyTemplate(TemplateView):

    name='Rakesh'

    template\_name='first/clsbsdtemplate.html'

    def get\_context\_data(self, \*args, \*\*kwargs):

        context = super(MyTemplate,self).get\_context\_data(\*args, \*\*kwargs)

        context['msg'] = 'variable value' # msg is context variable

context[‘var2’]=’some other value’

        return context

urls.py file

path('temp/',views.MyTemplate.as\_view(), name='template-view'),

**-:RedirectView:-**

This is mainly for used for redirecting to other url.

This view is available in 'django.views.generic.base' module

Example-1

Urls.py file

from django.views.generic.base import RedirectView

urlpatterns = [

    path('red/',RedirectView.as\_view(url='https://www.youtube.com/'), name='redirect\_view'),

]

Now if we try to access link for ‘red/’ url then it will take us to youtube

Example 2:

Views.py file

class MyRedirect(RedirectView):

    url='https://www.youtube.com/'

urls.py file

urlpatterns = [

    path('red1/',views.MyRedirect.as\_view(),name='myredirect'),

]

Now if we try to access link for ‘red1/’ url then it will take us to youtube

**-:Generic class based view:-**

django generic view are built of these base vies and were developed as a shortcut for common usage pattern such as displaying the details of an object.

These are **called Model related views**.

These of three types:

1. **Display View**
2. List View ------ to list/display all record of a model
3. Details view ---- to get details for a particular record
4. **Editing View**
5. Form view
6. Create View
7. Update view
8. Delete view
9. **Date View**
10. YearArchive View
11. MonthArchive view
12. WeekArchihve view

**Display View**

**List View**

Available in 'django.views.generic.list'. This is used when want to fetch all records for a model class.

If using list view then always need to specify then model name in our class (which have inherited ListView)

While this view is executing self.object\_list will contians list of objects(usually bit not neccerrarily a queryset) that the view is operating upon

**NOTE:**

* modelclass\_list.html--------->default temple name for ListView
* modelclass\_list or object\_list----------------> default name of context
* context variable ------ django quryset data type.
* django by default append '\_list' as suffix with model template name (model name), that's why template name becomes modelclass\_list.html
* If we don't want '\_list' to be suffixed then we can change it by 'template\_name\_suffix' variable.

e.g--

template\_name\_suffix='\_template' then template name will be searched by name 'modelclass\_template.html' name for that view class

**Example 1:**

Showing all record for Employee model using default variables (template name--- modelclass\_list.html context variable ---- modelclass\_list or object\_list)

views.py file

class ListCBV(ListView):

    model=Employee

models.py file

class Employee(models.Model):

    ename=models.CharField(max\_length=100)

    esal=models.FloatField()

    email=models.EmailField()

    eid=models.CharField(max\_length=10)

employee\_list.html

<body>

    <h3>Getting data using modelclass\_list </h3>

    {% for each in employee\_list %}

        {{each.ename}}

    {% endfor %}

    <h3>Getting data using object\_list </h3>

    {% for each in employee\_list %}

        {{each.ename}}

    {% endfor %}

</body>

**Changing template name of list view**

We can change templaet name by using variable 'template\_name' but still we can use default template name.

If both are given the it modifies template name will be on priority

e.g--

template\_name='myapp/home.html'

Now it will try to search for home.html template. if not found then it will check for detault template name(modelname\_list.html)

**Changing context variable name of list view**

We can change context variable name by using variable context\_object\_name.

If we change the context variable name then it will not look for default context variable name.

context\_object\_name=my\_context

**Advantages:**

If using ListView then **fetching of all records, passing context will be done by django**. We don’t need to pass them.

**Some methods of List View**

1. get\_queryset()
2. get\_context\_data(\*args,\*\*kwargs)

**How to create List View**

Create your own class and inherit ListView classs

Specify the name of model for which this class based view will use

Don’t need to render any template, creating context (if not using default)

Example 2:

For same model as above, get all records in which context name is mycontext and template name is showdata.html

Solutions:

Views.py file

class ListCBV(ListView):

    model=Employee

    template\_name='testapp/showdata.html'

    context\_object\_name='mycontext'

showdata.html

<body>

    <h3>Getting data using modelclass\_list </h3>

    {% for each in mycontext %}

        {{each.ename}}

    {% endfor %}

    <h3>Getting data using object\_list </h3>

    {% for each in mycontext %}

        {{each.ename}}

    {% endfor %}

</body>

##################################

# Detail View #

##################################

This is used for fetching data/details for any specific record. Available in django.views.generic.detail.

If using DetailView then we need to always specify the model name.

If using detail view then we need to specify the primary key for which data will be fetched.

**Note:**

default template name --- modelname\_detail.html

default context name ---- modelname or object

We need to specify he primary key in url mapping of urls.py file for which data will be fetched. e.g--

path('somepath/**<int:pk>**/',DetailViewClass.as\_view()), --- We can use any other word for pk.

Views.py file

class DetailViewCBV(DetailView):

    model=Employee

By default template name – employee\_detail.html and context name --- employee

Urls.py file

urlpatterns = [

    path('detailview/<int:pk>/',DetailViewCBV.as\_view()),

]

employee\_detail.html

<body>

    <h3>this is detailView CBV</h3>

    {{ employee.ename }}

    {{employee.esal}}

</body>

Changing the template name and context name:

If we want to change the template name and context variable name then we can change it using below two variables (same as List view)---

* template\_name= custom template name
* context\_object\_name= custom context name

**Question:**

Write DetailView for above model class.

Views.py file

class DetailViewCBV(DetailView):

    model=Employee

    template\_name='testapp/showdata.html'

    context\_object\_name='mycontext'

urls.py file

urlpatterns = [

    path('detailview/<int:pk>/',DetailViewCBV.as\_view()),

]

showdata.html

<body>

    <h3>Getting data using modelclass\_list </h3>

    {{mycontext.ename}}

    {{mycontext.esal}}

</body>

####################

# Create View #

####################

This view is used to create resources or add data in DB on specified model.

We need to specify the model name in inherited CreateView class and define get\_absolute\_url() in model class.

If we are using create view then we need to define **reverse url** using get\_absolute\_url() for model.

def get\_absolute\_url(self):

return reverse('url\_name ', kwargs=None, args=None)

OR

return reverse('app\_name:url\_name ', kwargs=None, args=None)

We can’t pass the kwargs and args in reverse at same time.

**Reverse url:**

Once objects are saved or deleted then django would try to route to some url, that url is called reverse url defined in model class.

**NOTE:**

We need to specify the fields of model class that we want to insert/add using template

default template class --- modelname\_form

default context variable -- form

Question

WAP for employee model to add employee object.

Models.py file

class Employee(models.Model):

    ename=models.CharField(max\_length=100)

    esal=models.FloatField()

    email=models.EmailField()

    eid=models.CharField(max\_length=10)

    def get\_absolute\_url(self):

        return reverse('home') #home is name given to any url, used for reverse url

OR

return reverse('testapp:home') # alternatively we can use this also

urls.py file

urlpatterns = [

    path('create/',CreateViewCBV.as\_view(),name='createCBV'),

    path('home/', homepage, name='home'),

]

Views.py file

class CreateViewCBV(CreateView):

    model=Employee

    fields=['ename','esal','eid'] #fields that want to add from template

###########################

# Update View #

###########################

Update class based view is used to update data of model name based on some key (primary key)

We need to define model and fields for which update operation will be performed.

We need to also define the reversr url.

Default template name will be modelname\_form.html

Views.py file

class UpdateEmployeeUpdateView(UpdateView):

    model=Employee

    fields=['ename','esal','eid'] # it means we need to update ename,esal, eid for Employee

urls.py file

urlpatterns = [

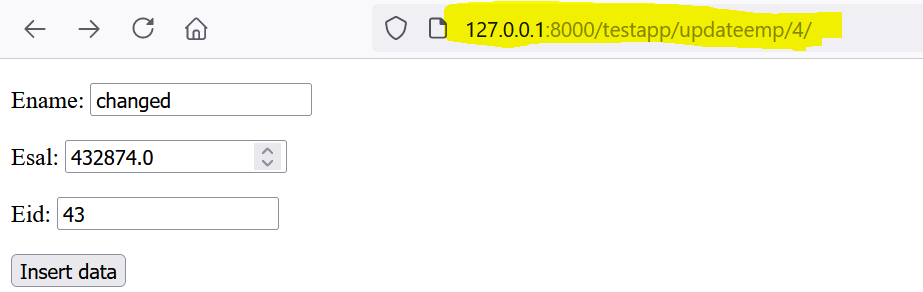
    path('updateemp/<int:pk>/',UpdateEmployeeUpdateView.as\_view(),name='updateemployee'),

]

**Reverse url:**

Once objects are created /updated then django would try to route to some url, that url is called reverse url defined in model class.

Output ( on hitting required url for updating data)



**Conclusion for model name and context name:**

ListView --- modelname\_list.html (templaet name) and modelname\_list (context variable)

DetailView --- modelname\_detail.html(templaet name) and modelname (context variable)

CreateView ---- modelname\_form.html, reverse url need to define using get\_absolute\_url

UpdateView ---- modelname\_form.html, reverse url need to define using get\_absolute\_url

########################################

# Delete View / Delete Operations #

########################################

It is used to delete records using class based view, delete operation is performed using primary key.

If using this view then we need to define the success\_url in view class, which route to specific page after delete operations.

Html/template page ----model\_confirm\_delete.html

context variable ----- model name

class DeleteCBV(DeleteView):

model=model\_class\_name

success\_url=reverse\_lazy('app\_name:url\_name')

**How it deletes data:**

First it renders the template file for deleting record. If it comes to same view from template then it deletes record and goes to template defined for success\_url.

Questions:

WAP to delete the record for below model class.

class Employee(models.Model):

    ename=models.CharField(max\_length=100)

    esal=models.FloatField()

    email=models.EmailField()

    eid=models.CharField(max\_length=10)

**Solutions**

Views.py file

class DeleteEmployee(DeleteView):

    model=Employee

    success\_url=reverse\_lazy('testapp:home')

urls.py file

urlpatterns = [

    path('delete/<int:pk>/',DeleteEmployee.as\_view(),name='deleteemp'),

    path('home/', homepage, name='home'),

]

Employee\_confirm\_delete.html

<body>

    <h3>this is delete page</h3>

    <h3>Are you sure you want to delete {{employee.ename}}</h3>

    <form action="" method="POST">

        {% csrf\_token %}

        <input type="submit" value="Delete">

    </form>

</body>

Note:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | List view | Detail view | Update view | Delete View | Create View |
| Default template name | ModelClass\_list.html | modelname\_detail.html | modelname\_form.html |  | modelname\_form.html |
| Default context object | Modelclass\_list  Or  object | modelname  or  object | define model and fields that want to update for given model |  | define model and fields that want to create for given model |
| Custom template name | template\_name=string | template\_name=string | Need to have get\_absolute\_url in model class |  | Need to have get\_absolute\_url in model class |
| Custom context name | context\_name =  string | context\_name =  string | Define model name and fields variable, enough for update operation |  | Define model name and fields variable, enough for create operation |
| template suffix name | template\_name\_suffix = string |  |  |  |  |

**Exception in dajngo**

Django have below package which can be used to handle exceptions.

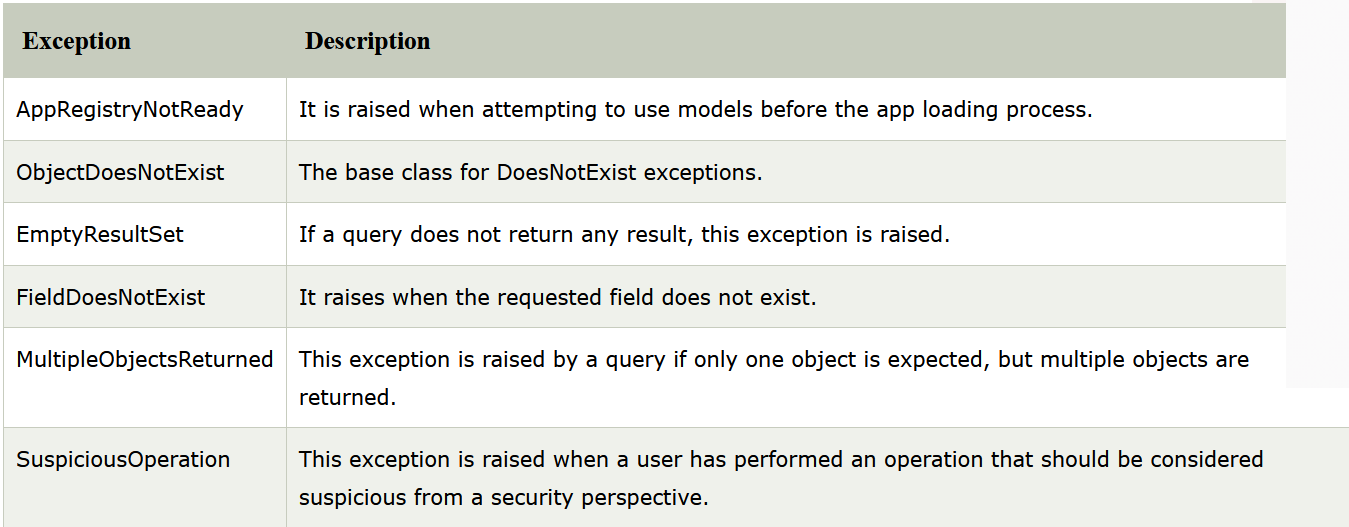
1. django.core.exceptions package
2. URL resolver Exception
3. Database Exception
4. Python Exception ----- all about core python exception
5. Testing Framework exception
6. Transaction Exception
7. Http Exception

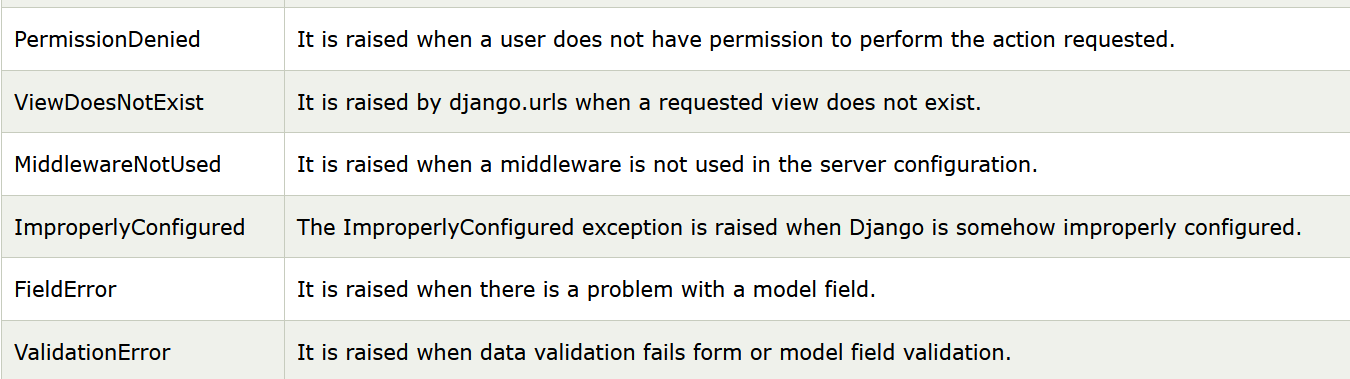
**django.core.exceptions package**

This module has use-cases like:

* When you are working on a custom middleware.
* When making some changes to Django ORM.

Th has below exception classes.





**ObjectDoesNotExist**

It is the base class for all the DoesNotExist Errors

ObjectDoesNotExist emerges mainly from get() in Django

**EmptyResultSet**

When we generate a query for objects and if the query doesn’t return any results, it raises this error.

**FieldDoesNotExist**

When a requested field does not exist in a model, this meta.get\_field() method raises this exception

**MultipleObjectsReturned**

When we expect a query to return a single response/ object but it returns more than one object, then Django raises this exception

**ViewDoesNotExist**

this is realated with wrongmapping of url and view

**ValidationError**

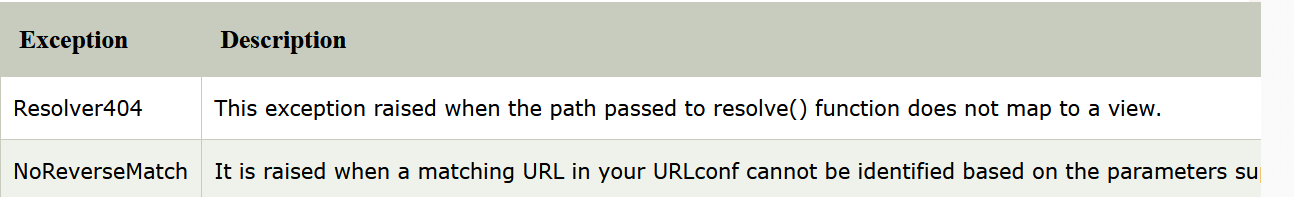
this is raised when data validation is failed for model or form.

It will match the data with the model field or forms field. And, it ensures that no security threat is there due to illegal characters in user-input.

**URL resolver Exception**

These exceptions are defined in django.urls module.

This class is a major part of urls.py for defining urls. We import our path function from urls class



**Resolver404**

We raise this exception if the path() doesn’t have a valid view to map

**NoReverseMatch**

It is a common occurrence. When we request for a URL which is not defined in our urls-config

**what is List Comprehension**

List comprehension is an elegant and concise way to create a new list from an existing list in Python.

*A list comprehension consists of an expression followed by for statement inside square brackets*

e.g –

pow2 = [2 \*\* x for x in range(10)]

print(pow2)

#################################################################################### **STATIC FILES** #

###################################################################################

We can use static files in two places—

1.) Static files inside project

2.) Static files in application

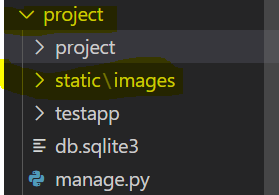
####################################

# Static files (images,videos etc) in Project #

####################################

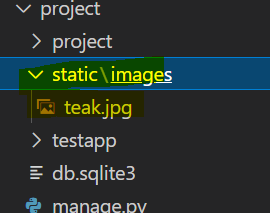
If we are using static files inside our project then we need to instruct Django to look for static files in project folder/directory. This is done by using 'STATICFILES\_DIRS' variable, it’s list type.

STATICFILES\_DIRS=[os.path.join(BASE\_DIR,'static'),] --------- Now Django will look in project level for static files.



Questions

Display a picture in a template where picture is present in static folder of project level as shown below.



Solutions

Urls.py file of project

from django.contrib import admin

from django.urls import path

from testapp import views

urlpatterns = [

    path('admin/', admin.site.urls),

    path('home/',views.home,name='home'),

]

Views.py file

def home(request):

    return render(request,'testapp/index.html')

index.html page

<!DOCTYPE html>

{% load static %}

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h3>this is index.html page</h3>

    <img src="{% static 'images/teak.jpg' %}" alt="alternate">

</body>

</html>

**How static tag works-**

1. First checked for static files as how it is instructed in setting.py files(inside project)
2. If not found as per instruction of setting.spy file then it checks for default path (inside app)

**Note:**

We can use static files from project or app in same project it all depends on searching criteria defined in setting.spy file i.e if we can define to use static files from project still, we can use from app.

#############################

# Static files (js,css etc) in Project #

#############################

If using css, javascript files then we need to link them in head section of templates.

<link rel="stylesheet" href="{% static 'relative\_path\_of\_static\_files' %}">

Example. In same example

Index.html

<!DOCTYPE html>

{% load static %}

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <link rel="stylesheet" href="{% static 'css/style.css' %}">

    <title>Document</title>

</head>

<body>

    <h3>this is index.html page</h3>

    <img src="{% static 'images/teak1.jpg' %}" alt="alternate">

</body>

</html>

style.css

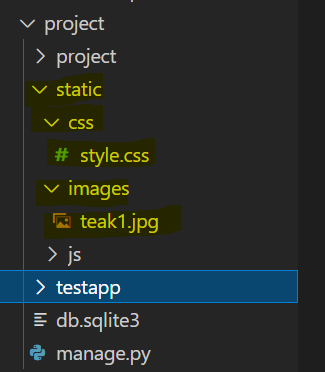
h3{

    color: red;

    background-color: yellow;

}

Project structure



##########################

# Static files inside application#

##########################

This we already know how to use it, nothing is required to update/add in setting.py file, default content will work.

1. If using javascript,css files in templates then we will have to link them exactly how we did in above example.

<link rel="stylesheet" href="{% static 'relative\_path\_of\_static\_files' %}">

#################################################################################### **Custom 404 handler or Page not found #** ###################################################################################

Suppose we want to handle page/url not found error in Django so that end user doesn’t know all our view business logic that add security to our application.

To handle page/url not found error we need to use handler404 variable in our projects urls.py file. It’s value is the function to handle the invalid url and create urls for invalid urls.

handler404='function\_or\_class\_for\_404\_handler' ---- syntax to sue it

path('^$',view\_for\_invalid\_urls,name='pagenotfound') --------- create urls for invalid urls in proj. urls

handler404='testapp.views.handler404' -------- # app\_name.views.handler\_view.name

Note:

1. If we use handler404 then it will handle any invalid url pattern.
2. If using handler404 then we need to set Debug= Flase and ALLOWED\_HOSTS in settings.py file i.e applicable for production environment.

Example

View.py file

def home(request):

    return render(request,'testapp/index.html')

def handler404(request,\*args,\*\*kargs):

    return render(request,'testapp/404.html')

def test(request):

    return HttpResponse('this is applications view')

urls.py file (project)

from testapp import views

import testapp

urlpatterns = [

    path('admin/', admin.site.urls),

    path('home/',views.home,name='home'),

    path('app/',include('testapp.urls')),

    path('^$',views.handler404,name='pagenotfound')

]

handler404='testapp.views.handler404' #app\_name.views.handler\_fuction

urls.py file (app)

import testapp

urlpatterns = [

    path('test/',views.test),

]

404.html (template name)

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Document</title>

</head>

<body>

    <h1>Sorry, Page not found</h1>

</body>

</html>

<http://127.0.0.1:8000/343> ---- on this it will render 404.html page as handler will be called b/c not valid url

<http://127.0.0.1:8000/app/758934> ---- on this it will render 404.html page as handler will be called b/c not valid url

**Note:**

You can see it is able to handle all type of invalid url doesn’t matter unmatch happening in app label.

###################################################################################

# **Using For loop in Django template** #

###################################################################################

**Passing multiple dictionaries in template as context**

We can pass multiple dictionary as context by creating another dictionary of them.

def home(request):

    cont={'name':'Rakesh','age':30,'my\_list':[1,23,22,12]}

    my\_dict={'d1':[1,23,22,12],'d2':[222,222,333]}

    form={'fm':cont,'prod':my\_dict}

    return render(request,'testapp/index.html',form)

here cont and my\_dict are dictionary variable, we are passing them as context after creating a new dictionary as form.

Note:

form={fm:cont,'prod':my\_dict}

Now to access value for from context we have to use prod and fm variable in our template.

**Accessing value in template (specific to a dictionary 'prod' )**

Accessing values of variable in template

Let’s say we want to access value specific to any dictionary(prod)

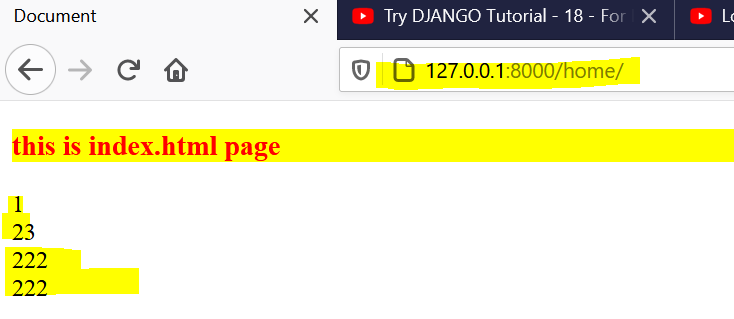
{% for key, value in prod.items %}

        <li>{{ value.0 }}</li>

        <li>{{ value.1 }}</li>

{% endfor %}

Output



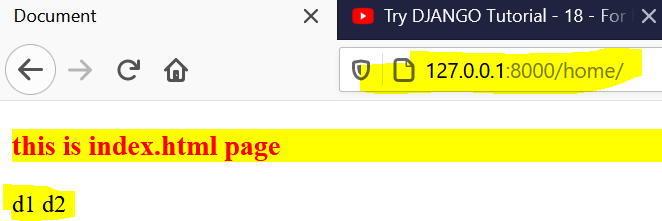
**#Accessing keys in template (specific to a dictionary 'prod')#**

{% for key, value in prod.items %}

        {{ key }}

{% endfor %}

Output



**Note:**

When we are using more than one dictionary as context then we need to access value of them one by one or specific to that dictionary.

Views.py file

# Create your views here.

def home(request):

    cont={'name':'Rakesh','age':30,'my\_list':[1,23,22,12]}

    my\_dict={'d1':[1,23,22,12],'d2':[222,222,333]}

    form={'form':cont,'prod':my\_dict}

    return render(request,'testapp/index.html',form)

urls.py file (project label)

urlpatterns = [

    path('admin/', admin.site.urls),

    path('home/',views.home,name='home'),

]