**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/#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).

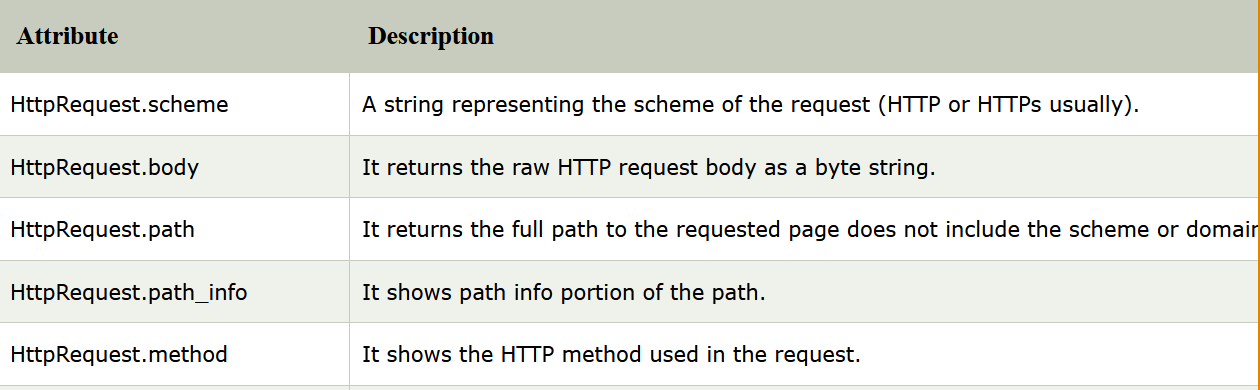
1. **Options.default\_permissions<permissions>**
2. **Options.constraints=<list of constrainsts>**

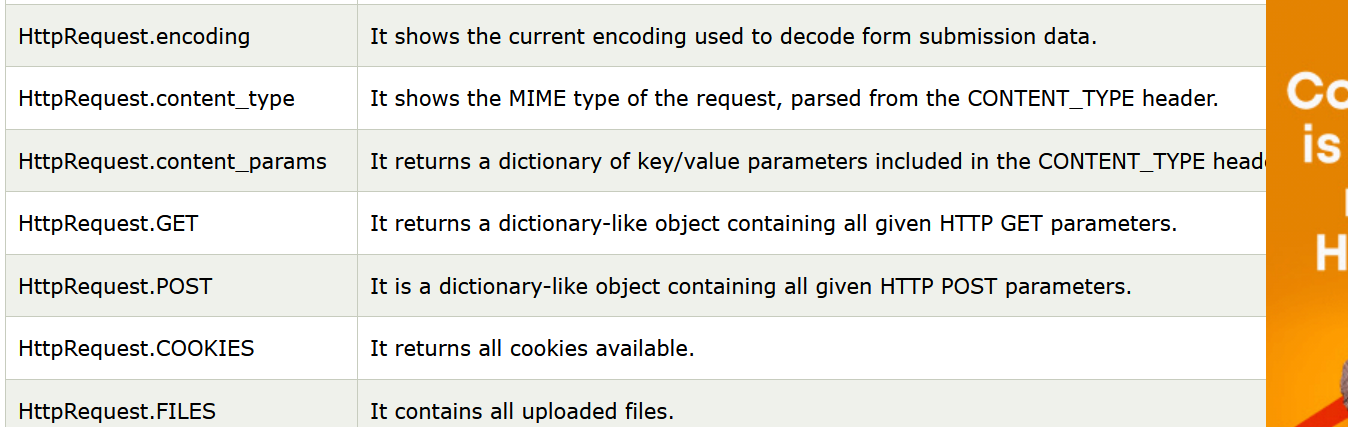
A list of constraints that you want to define on the model

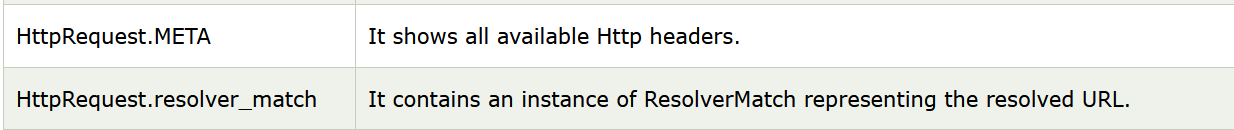
**Adding data to DB**



**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 mjkiactually 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 acending order
2. -**col\_name**----- then order/sorts in decending 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. **al()** ---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. **value\_list(\*field,flat=False,names=False)**

Returns values as tuple form instead of dictionary

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.

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**



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

This querysetevaluates to



1. **OR**

Combines two QuerySets using the SQL OR operator.

**qs1 OR qs2**



Above queryset evaluates to



**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 masny 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.

***Question***

Save/add data to student table(already created)

***Using save()***

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

s.save()

***Using create***

Student.create(=’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.

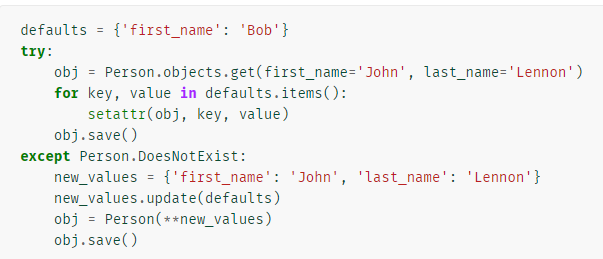
1. **update\_or\_create(defaults=None, \*\*kwargs)/update\_or\_create()**

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.

Let us take example of create or update manually.



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



1. **bulk\_update()/bulk\_update(objs, 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()**
10. **bulk\_create(objs, batch\_size=None, ignore\_conflicts=False)**

**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

**Relationships in Django**

We learn relationship in Django. It has two type

1. Many-to-One
2. Many-to-Many
3. One to One
4. **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

**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 table to create for storing the many-to-many data

If not used then Django will use table1\_table2 name as intermediatery 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)*

**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. **Directory 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 }}

{% elifathlete\_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')*

**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



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**
2. **label="custom label"**
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**
2. forms.Widgets handling input of text
3. forms.NumberInput
4. forms.EmailInput
5. forms.URLInput
6. forms.PasswordInput
7. forms.Textarea
8. **Selector and checkbox widgets**
9. forms.CheckboxInput
10. forms.select

etc.

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

**-: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.

{% 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>

**-: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).

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

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

**Creating cookies**

**HttpResponse.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

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*

**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

**Deleting the cookie**

**HttpResponse.delet\_cookie(kay,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

**-: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

**Creating/reading/modifying the sessions**

**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 -- retunrns 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.erequest.session.keys()*

***Some useful methods of sessions***

**request.get\_session\_cookie\_age()---** gives the age of session cokkies. 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.

**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 defaul age of cookie
2. **SESSION-COOKIE\_DOMAIN**--- the domina to use for session cookies
3. **SESSION\_CACHE\_ALIAS**---used if using the chached 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 s 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

**-:CACHE:-**

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

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

Django supports three type of caching.

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

**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.

These 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 inyou 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.*

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, 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 inyou 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\_saveetc
2. **custom signals** ---

**Note:**

For creation 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*

@receiver(post\_save, sender=MyModel)

defmy\_function\_post\_save(sender,\*\*kwargs):

1. **Using connect()**

Use the signals function 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**

**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,request,user,\*\*kwargs):**

**#**

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

**#**

sender----- signal sender

request--- HttpRequest object

user--- django user object

\*\*kwargs--- keyword arguments

defmy\_function\_post\_save(sender,\*\*kwargs):

# do the action…

**CUSTOM SIGNALS**

**Steps:**

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

Create the signal receiver

Bind the signal

Send the signal

**Creating signal**

class Signal(providing\_args=list)

theproviding\_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()

Example

Signals.py file in app

#creating notification signal

from django.dispatch import Signal, receiver

notofication=Signal(providing\_args=['rakesh','my\_nm'])

@receiver(notofication)

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')

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

**-: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

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



**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 got 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’ ion 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**.

**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 I 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 owen 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 but that using parameters/value in as\_view() but all those arguments must be available 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.

**-: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 directly from urls.py file, 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'),

]

This way of writing doesn’t needs MyTemplate class to be present in views.py file

**Passing context to TemplateView**

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,self).get\_context\_data(\*args, \*\*kwargs)

context['name'] = 'Gryffindor'

context['var'] = 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'] = self.name

        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 off for of thise base vies and were developed as a shortcut for common usage pattern such as displaying the details of an object.

These of three types:

1. **Display View**
2. List View
3. details view
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'

A page representing a list of objects.

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---------> name of temple for ListView
* modelclass\_list or object\_list----------------> name of context
* django by default append '\_list' as suffix with mode templaet 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

**Changing template name of list view**

We can change templaet name by using variable 'templa\_name' but still we can use default template name.

If both are given the it modified 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

**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

Let’s take an example of creating a ListView

**Exception in dajngo**

Django have below package which can be used to handle exceptions.

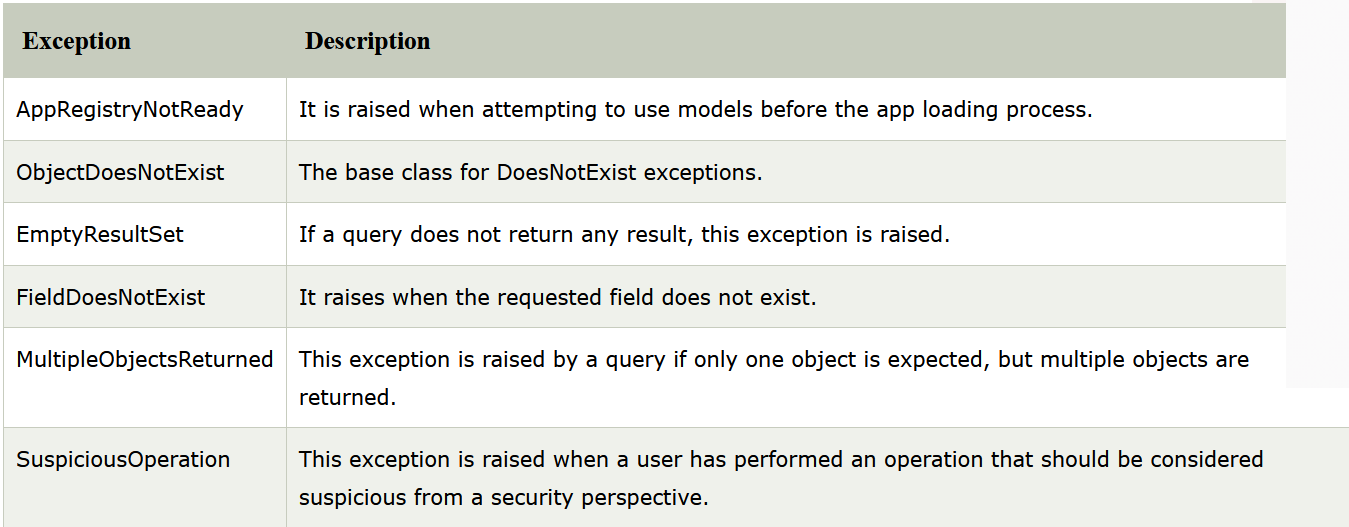
1. django.core.exceptions package
2. URL resolver Exception
3. Database Exception
4. Python Exception ----- all about core python exception
5. Testing Framework exception
6. Transaction Exception
7. Http Exception

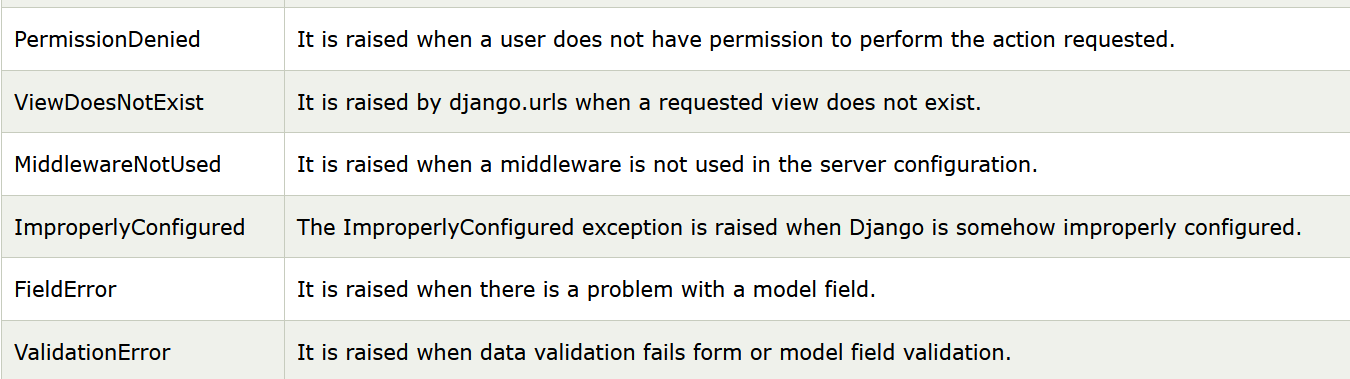
**django.core.exceptions package**

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**

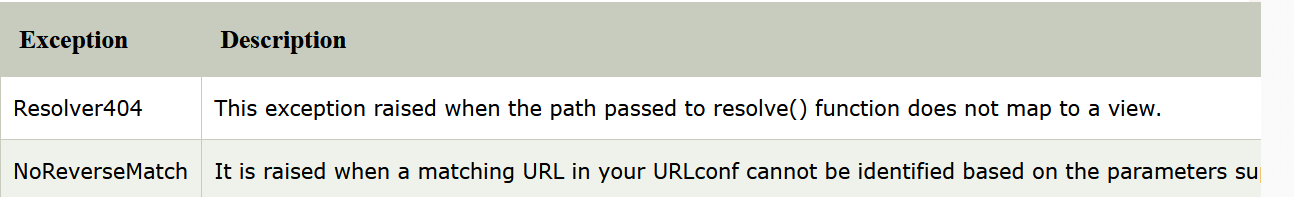
this is raised when data validation is failed for model or form.

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)