Django Rest API

NARAYANA SIR

/* Displaying a simple message in Django project */

=> Goto views.py, file & write below code;

=> from django.http HttpResponse

```
def home(request):
    return HttpResponse("Hello Django")
```

--> Goto urls.py file & write below code;

=> from institute import views

```
urlpatterns= [
        url('_____'),
        url(r'home', views.home)
    ]
```

/* Displaying a multiple views message */

=> Goto views.py file & write below code;

=> from django.http import HttpResponse

```
def home(request):
    return HttpResponse("This is my first program")

def contact(request):
    return HttpResponse("please go to main office")

def location(request):
    return HttpResponse("Maithivannam Building")
```

--> goto urls.py file, & write below code;

=> from courses import views

```
urlpatterns=[
    url(r'^home1', views.home),
    url(r'^contact1', views.contact)
```

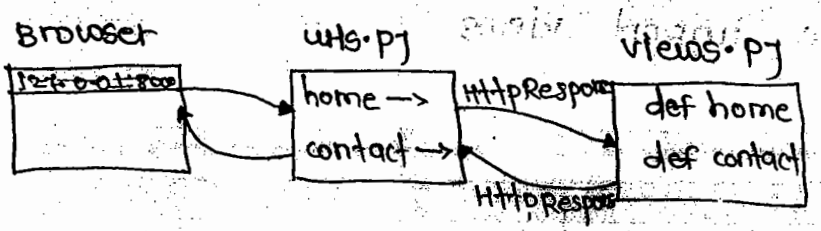
```
    url(r'location/', views.location),
]
```

—> copy the Ipaddress & paste on urlpath of a browsers.

=> https://127.0.0.1:8000/home/

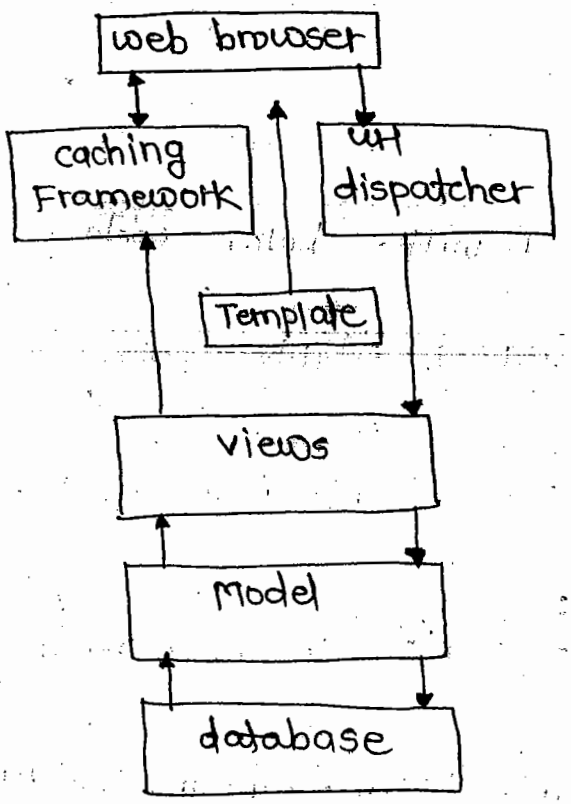
* working Flow:

=>



* Architecture of Django Framework:

=>



—> Django Framework follows MVT design patterns;

=> MVT [model views Template]

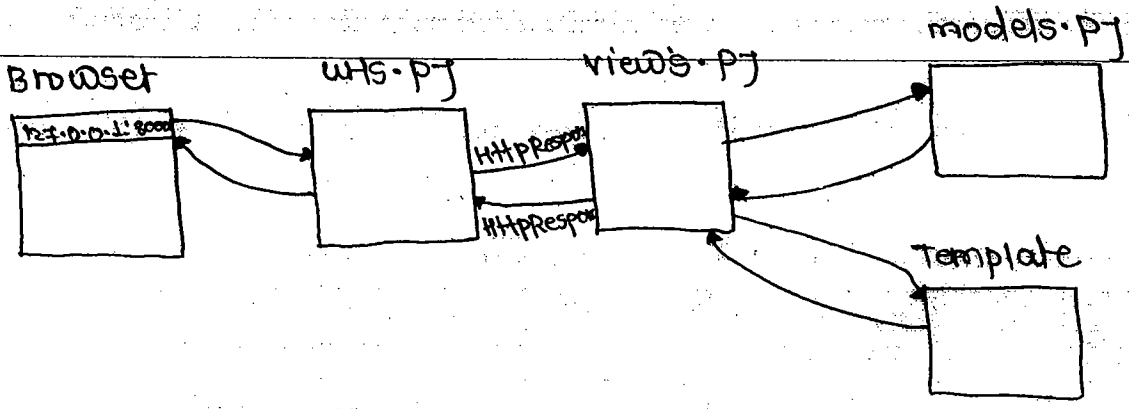
* M - It is used to database coding.
* views - It is used to write a buyiness logic.
* Template - It is used to write presentation code.

⟹ MVC [Model views controller]

* M – It is used for database access or manage it.
* V – It is used for presentation code.
* C – It is used for Business logic.

* working Flow :

⟹



⟹

① --init--.py : It represents that, the current directory belongs to django project.

② settings.py : It is used to write configure a setting of a django project & applications.

③ urls.py : It is used to gives the mapping between urls & views files.

④ wsgi.py : (webserver gateway interface), It is useful in a running a server.

⑤ manage.py : It is also called as command line utility, which is mainly used for a administrative purpose.

- It is a python file & contains lot of commands;

⟹ migrate.

⟹ makemigrations.

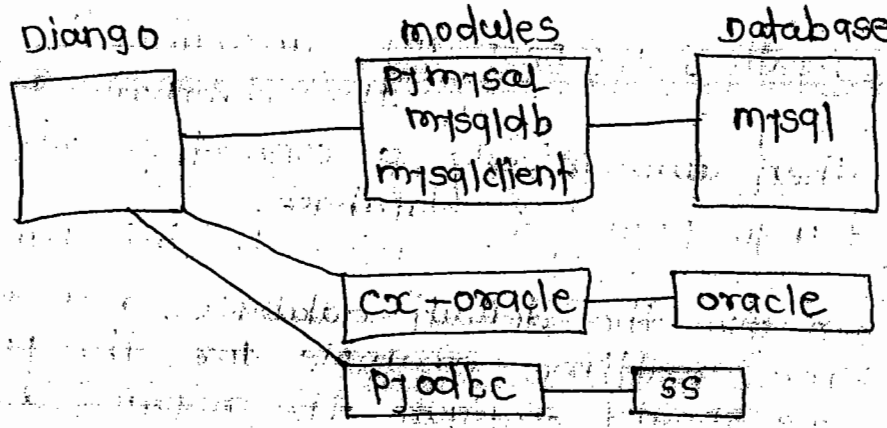
⟹ createsuperuser.

⟹ runserver.

* configuring Database: we can configure any database in django framework, but generally we configure, we use a mysql, oracle, postgresql, etc.

→ Every database has it's own module, which is required to configure the database, in a django project/applications

⇒



→ we should download & install, corresponding module in commond prompt by using "pip" command.

✱→ Creating a database;

→ open mysql server (mysql command line) & create a database.

⇒ create database modeldb;

⇒ use modeldb; (database changed)

⇒ show tables;

→ goto setting.py file & configure the database;

⇒
```
Databases = [
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'modeldb',
        'USER': 'root',
        'PASSWORD': 'root',
    }
]
```

- By default, sqlite3 has two parameters, ENGINE [servermang

and Name [database name].

→ If we configure any other database, then we required some other parameters also, password, or port / Host.

```
"""
DATABASES = {
        'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
```

→ Whenever we specify or configure the database in setting.py file, then automatically a connection will be created between django project & database.

→ If we want to create which is going to into production server, then we can use the default database, & if we want to create some realtime example for the project, which are going to, we should configure the required database.

* Verifying connections: We can verify the database details, which are configure the setting.py file.

→ we execute a connections by using cursor(), if it will return any error, then we configure the database property in a setting file.

→ If we didn't return any error then we configure properly.

→ we should goto python shell to verify the connections.

⇒ python manage.py shell

⇒ from django.db import connection

>>> q= connection.cursor()

here, a cursor(), did not return any error, that means we configure a database properly, in setting.py

→ If we configure, a database properly, then we can create a models, in models.py file.

* Creating a Models: A model is a python class, which contains, field-name & fieldtype.

→ models are used to manage a data in the database
→ If we execute, these models then model name becomes as a table name, field name as, columns, & fieldtype becomes as a datatypes.

* Inserting a Data: we can insert the data to the table in a different ways;

① Through python shell.
② Through database.
③ Through adminsite.

→ we can perform all operations in the above three ways;
→ we will perform all operations through admin site after discussing admin site.
→ we can insert a data in a two different ways;

① save(): If we take the data in the object, then we should save any, by using save() method.
→ If we don't save the object by using save(), then the data, will not insert into the database.
→ we should goto python shell to perform all operations;

⇒ python manage.py shell
⇒⇒ from modelapp.models import Emp
⇒⇒⇒ a=Emp(eno=10, ename='vikas', sal=50,000)
>>> a.save()
>>> b=Emp(eno=11, ename='Raj', sal=60,000)

② create(): In this method, we don't require to save the object by using save(), beacuse the create() method, to save the data "automatically" to saved in the database.

⇒⇒⇒ c=Emp.objects.create(eno=12, ename="Ram", sal=30,000)
d=Emp.objects.create(eno=13, ename="Ravi" sal=20,000)

→ go to the database & check the data in database.

⇒ select * from modelapp_emp;

② Through database: when we want to create a database in mysql server then, we should run the following query;

⇒ Insert into modelapp_emp(eno, ename, sal) values (14, 'siva', 10,000);

>>> insert into modelapp_emp(eno, ename, sal) values (15, 'yuvi', '20,000');

⇒ select * from modelapp_emp;

⇒

id	eno	ename	sal
14	14	siva	10,000
15	15	yuvi	20,000

* Filtering the Data: we can filter the data through python shell & also through database, using a filter() & Exclude().

① Through python shell: If we want to filter the data through python shell, then django supports two different ways;

*① Filter(): This method will retrieve the data as per the given "keyword_arguments", in the filter() method, whatever the Argument specified, the filter(), the corresponding data only, will display in the output only.

⇒ >>> x = Emp.objects.filter(sal=20,000)

>>> print(x)

[<Emp: yuvi>]

```
>>> y = Emp.objects.filter(eno=13)
    print(x)
  [<Emp: ram>]
```

*② Exclude: It will retrieve the data except the data which belongs to the specified a "keyword_argument" in a exclude() method.

```
=>>> x = Emp.objects.exclude(eno=13)
    print(x)
[<Emp: vikas>, <Emp: Ravi>, <Emp: Ram>]
```

② Through python shell:

```
=>>> select * from modelapp_emp where eno=15;
```

=>

id	eno	ename	sal
3	15	satya	20,000

→ If we run; filter(eno=11), in the python shell, then it will convert like, where eno=11.

```
→ select ename from modelapp_emp where eno=12;
```

=>

ename
vikas
Ravi
yuvi

→ If we run: exclude(eno=11), in the python shell, then it will convert like, where eno=11.

* Retrieving the data : (fetching a data)

*① Through python shell: we can retrieve the specific value or entire employee names.

```
>>> a.ename 'nani'
    a.sal 10000
    b.email 'satya@gmail.com'
    c.eno 12.
```

• NOTE: If we don't create a string representation in model then it will not display name.

→ It is displaying only ename because, we specified only ename in string

→ If we use, all() method to get a all employee names from the database table.

```
>>> x = Emp.objects.all()
    print(x)
[<Emp: vikas>, <Emp: Ram>, <Emp: Ravi>, <Emp: yuvi>]
```

*② Through Database :

```
>>> select * from modelapp_emp;
```

→ It will displays all employee data.

* Updating a Data : we can update the data in a python shell & also in a database

*① Through python shell : we updating for the data in a python shell & also in a database,

→ we updating for the save object, if we don't save that we will not modify the data in the database

```
>>> a.sal 10000
```

→ To update the a.sal;

```
>>> a.sal = 15000
    a.save()
>>> d.email 'ram@gmail.com'
```

→ To update the d.gmail.

```
>>> d.email = 'ram123'@gmail.com'
    d.save()
```

• NOTE: Now we can see the modified data in the database, for all this modifications.

* Through Database: we use a update command to update the table data;

==>>> update modelapp-emp set sal = 50000 where ename='siva';

==>>> select * from modelapp-emp;

* Deleting Data: we can deleting data through python shell & through database.

*① python shell: In this, we use a get() method to get a specific record into an object.

→ we use a delete() method, to delete a object.

•NOTE: Now check the table for modifications.

```
==>>> x=Emp.objects.get(eno=10)
      x.delete()
  (1, {'modelapp.Emp': 1})
```

*② Through Database: we use a delete command to delete a specific employee data.

→ we should specify a specific filter data in a where conditions.

==>>> delete from modelapp-emp where eno=11;

* Manager: A manager is a Interface, which provides a communication between the model & a corresponding methods.

→ Each model must have a manager bydefault.

→ The Default manager name is objects.

→ we can change the manager name by using a manager field.

→ If we change the manager, then we should use a user-defined manager only, if we try to use a default name, the django returns a "attribute_errors".

→ goto models.py file & create a models with a manager fields.

```
=> from django.db import models
   class Emp(models.Model):
       eno = models.IntegerField()
       ename = models.CharField(max_length=20)
       esal = models.IntegerField()
       email = models.EmailField(max_length=30)
       obj1 = models.Manager()
              def __str__(self):
                  return self.ename
```

—> Now, Run the commands ;

```
=> python manage.py makemigrations.
=> python manage.py migrate.
=> python manage.py createsuperuser.
=> python manage.py runserver.
```

—> run shell commands, to goto python shell ;

```
=> python manage.py shell.
=> from modeldbapp.models import Emp.
=>>> a = Emp.objects.create(eno=10, ename='vikas', sal=30,000)
=>>> x = Emp.obj1.create(eno=11, ename='Ram', sal=20,000)
     x = Emp.obj1.filter(eno=10)
     print(x)
=> [eno=11 <nani>]
```

* Model Relationship : Django models follows RDBMS - (Relational database mgmt systems).

—> generally, we use a relationships "To avoide a duplicate" / "unwanted data" from the models (tables).

—> Django supports three (3) types of Relationship ;

① one to one relationship : In one to one Relationship child Record, depends on only one parent record.

—> we use a "one to one-field", to implement one to one relationship.

—> If, we try to give multiple child records for same parent record, then it will throw an errors.

—> whenever, we use a one to one model, then that model called as child models, & whatever modelname, we use as a argument in onetooneField(), that is called as parent model.

⇒

—> project name : otopro.

—> App name : otoapp.

—> db name : otodpmdb.

—> open pycharm with otopro project folder.

—> goto setting.py file & configure database & Installed-apps;

⇒ INSTALLED_APPS=[

:

'otoapp',

]

⇒ DATABASES={

'default' : {

'ENGINE' : ------.Mysql',

'NAME' : 'otodpmdb',

'USER' : 'root',

'PASSWORD' : 'root',

→ goto models.py file & write below code;

```
⇒ class student(models.Model):
      sno = models.IntegerField()
      sname = models.charField(max_length=20)
      sloc = models.charField(max_length=40)
      def __str__(self):
          return self.sname

  class course(models.Model):
      student = models.OneTooneField(Student, on_delete=models.
                                                    CASCADE)
      cno = models.IntegerField()
      cname = models.charField(max_length=40)
      cfee = models.IntegerField()
      def __str__(self):
          return self.cname
```

→ goto admin.py file & write below code;

```
⇒ from .models import student, course
    class Adminstudent(admin.ModelAdmin):
        list_display = ['sno', 'sname', 'sloc']
    class Admincourse(admin.ModelAdmin):
        list_display = ['cno', 'cname', 'cfee']

    admin.site.register(student, Adminstudent)
    admin.site.register(course, Admincourse)
```

→ To Run the commands;

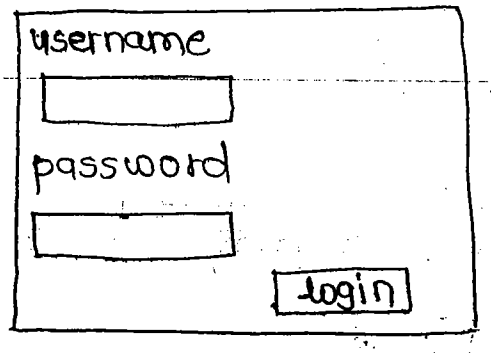
⇒ python manage.py makemigrations.
⇒ python manage.py migrate.
⇒ python manage.py createsuperuser.
⇒ python manage.py runserver.

→ copy the Ipaddress & open the browser & paste the Ip address on urlpath along with /admin.

⇒ [http://127.0.0.1:8000/admin/]

→ then, it open's admin login page.

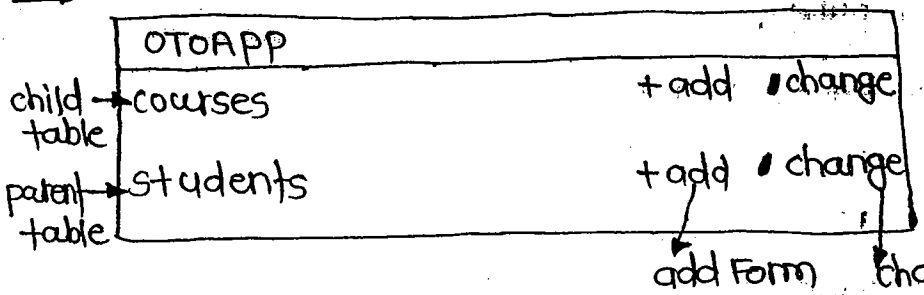
⇒ Django administrations



→ If we click on login button, it will open admin home page.

→ we can see our models, in a admin home page.

⇒

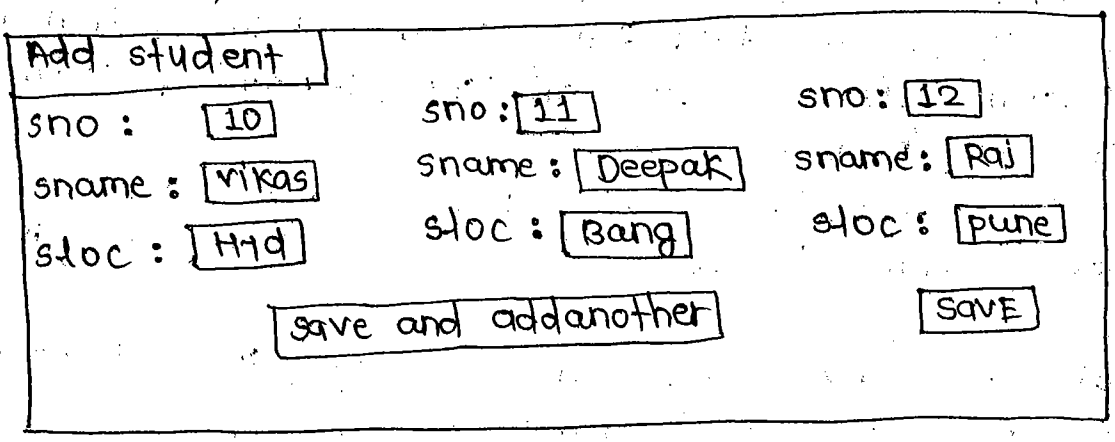


→ AddForm is used to add new data, in existing table.

→ change list is used to display / modify the existing data.

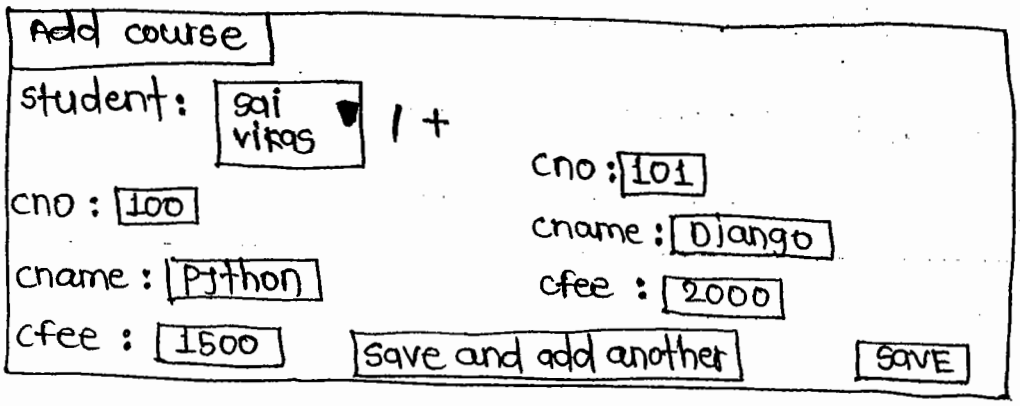
→ click on Add form of student model & give the data like below;

⇒



→ click on add form on coure models.

⟹



→ now goto database & check the database;

⟹ select * from otoapp_student;

⟹

id	sno	sname	sloc
1	100	vikas	Hyd
2	102	Deepak	Bang
3	103	Raj	pune

→ select * from otoapp_course;

⟹

id	cno	cname	cfee	student_id
1	100	Python	1500	1
2	101	Django	2000	2

→ All courses are depend on individual parent record, beacuse we takes only onetoone Fields.

here, student table is called as parent-table & course table, called child-table.

→ In parent table, parent 1 (id=1), parent 2 (id=2) are having children in child table (course_table), that means 'Python' course is depending on 'sai' student & 'Django' is depending on 'Deepak' student.

- generally, we can not delete 'sai' record & Deepak record, beacuse they have dependency, in child table.

→ but, we can delete, ram student, from student, beacuse, ram is not Learning any course.

→ if we want delete, a parent records (sai & Deepak) then corresponding child records, can be managed by using "cascading-rules".

* cascadeing-rules: It is used to manage child table, when we delete a corresponding parent records.

① DO-NOTHING: If we set on-delete = models.DO-NOTHING, [NO-ACTION] in the student of course model, then we can't delete the parent records (sai & Deepak), or update the parent records, which are having child records.

→ goto models.py, and modify studentField of course model, like below;

⇒ student = models.oneToOneField(student, on_delete= models.DO-NOTHING

→ run the commands; (all)

→ runserver

→ goto admin site, & refresh admin site.

→ click on change list student model, & goto student, model

→ select sai record & try to delete sai record;

⇒ yes, Iam sure.

→ It will return error, beacuse we can not delete "sai" record. beacuse, we used DO-NOTHING used here.

② set-NULL: If we set, on-delete = models.SET-NULL, in student field of course model then, it allows us to delete parent records, but the corresponding child us to delete parent records, but the corresponding record id, will become 'NULL'.

- generally, by default all columns will not allow null values, so here, we also set null = True, beacuse child will be become as null values.

→ goto models.py file & modify the student Field of course models, like below;

⇒ student = models.oneToOneField(student, on_delete= models.SET-NULL, null = True)

→ run the comands; (all)

→ runserver.
→ goto admin site, & refresh admin site.
→ click on change list of student model.
→ select sai record and delete sai record.
→ now goto database & check the tables.

⇒ select * from otoapp_student;

⇒

id	sno	sname	sloc
2	11	satya	Bang
3	12	nani	pune

⇒ select * from otoapp_course;

⇒

id	cno	cname	cfee	student_id
1	100	Python	1500	NULL
2	1001	Django	2000	2

③ Set(value): If we set, on_delete=models.SET(value), then, it allows to delete the parent record & it replace, this new value in the place deleted parent record id, then the corresponding student id (3) will be deleted.

→ goto models.py & write some code, or courses & modify student field like below;

⇒ student = models.OneToOneField(student, on_delete = models.SET(3), null=True)

→ run the comands; (all)
⇒ runserver
→ goto admin site, & refresh admin site.
→ goto student model & delete a satya record.
→ goto database & check the tables;
⇒ select * from otoapp_student;

⇒

id	sno	sname	stoc
3	12	nani	pune

⇒ select * from otoapp-course;

⇒

id	cno	cname	cfee	student-id
1	100	Python	1500	Null
2	101	Django	2000	3

④ CASCADE : If we set on_delete = models.CASCADE, then it will delete the parent record & it also delete a corresponding child record.

→ goto models.py & modify the student field in the course model, & like below;

⇒ student = models.OneToOneField(student, on_delete = models.CASCADE).

→ run the commands again; (all)
→ runserver.
→ goto admin site & refresh it;
→ goto student models, & Delete all nani records;
→ goto database & check the tables;

⇒ select * from otoapp-student;
→ empty set [Deleted]

⇒ select * from otoapp-course;

⇒

id	cno	cname	cfee	student-id
1	100	python	1500	NULL

② many to one relationship: In this relationship multiple child records can depends on 'single-parent' records.

—> we use a 'Foreign key" to implements ,, many to many relationship.

—> A single student can learn multiple courses, this means, it follows 'many to one" relationship.

—> we use 'foreign key', beacuse it allows multiple duplicate values also.

⟹ project name: Mtoapp.

—> App name : Mtoapp.

—> db name: mtoapmdb.

—> goto setting.pj file, & write or configure database & Installed_apps;

—> goto models.pj file & write below code;

```
⟹ class Student (models.Model):
        sno =models.IntegerField()
        sname= models.charfield (max_length=20)
        sloc = models.charfield (max_length=30)
        def __str__(self):
            return self.sname

    class course(models.Model):
        student = models.ForeignKey (Student, on_delete = models.CASCADE)
        cno = models.IntegerField()
        cname= models.charField (max_Length=40)
        cfee = models.IntegerField()
        def __str__(self):
            return self.cname
```

—> goto admin.pj, & write below code;

```
=> from .models import Student
   class AdminStudent(models.ModelAdmin):
        list_display ['sno', 'sname', 'sloc']
```

```
class Admincourse(admin.ModelAdmin):
        list_display ['cno', 'cname', 'cfee']

admin.site.register(student, Adminstudent)
admin.site.register(course, Admincourse)
```

—> to run the commands ; (all)
—> runserver . [copy IP address & paste] along with /admin
—> goto admin site & refresh it. [username & password]
—> you can see your both models , in a admin home page;

=>

MTOapp	
cources	+add / change
students	+add / change

—> click on add Form of student table ; & gives the data , like below ;

=>

Add student.

sno : 10	sno : 11	sno : 12
sname : sai	sname : satya	sname : nani
stoc : Hyd	stoc : Bang	stoc : pune

save & add another | save

—> click on course model, of add Form & give the data;

=>

Add course

student [sai ▼] 1+

cno : 101	cno : 102	cno : 103
cname : Python	cname : Django	cname : sql
cfee : 1500	cfee : 2000	cfee : 500

→ goto database & check the tables;

⟹ select * from mtoapp_student;

⟹

id	sno	sname	sloc
1	10	sai	Hyd
2	11	satya	Bang
3	12	nani	pune

⟹ select * from mtoapp_course;

⟹

id	cno	cname	cfee	student_id
1	101	python	1500	1
2	102	Django	2000	1
3	103	sql	500	2

→ here, student table is called as parent-table, & course table is called a child table.

Both, python & Django courses, are learning by a single student ('sai') that means multiple courses learning by a single student, that's why, it is called as "many-to-one" relationship.

here, we can also use all cascading rules, to manage the child records, when we delete a parent records, or corresponding parent records.

③ Many to Many relationship: In this relationship a multiple, child records can depends on multiple parent records.

→ we use a "manytomanyField" to implements a many to many relationship.

→ It supports only "CASCADE" rules.

→ multiple courses learn by multiple students.

⟹ project name: mtmpro.

→ app name: mtmapp.

→ db name: mtmsprmdb.

→ goto setting.py file & configure database & Installed_apps ;

→ goto models.py & write below code;

=> class student(models.Model):

```
    sno = models.IntegerField()
    sname = models.charField(max_length=20)
    sloc = models.charField(max_length=40)
    def __str__(self):
        return self.sname

class course(models.Model):
    student = models.ManyToManyField(student)
    cno = models.IntegerField()
    cname = models.charField(max_length=20)
    cfee = models.IntegerField()
    def __str__(self):
        return self.cname
```

→ goto admin.py & write below code;

=> from .models import student, course

```
class Adminstudent(admin.ModelAdmin):
    list_display ['sno', 'sname', 'sloc']
class Admincourse(admin.ModelAdmin):
    list_display ['cno', 'cname', 'cfee']

admin.site.register(Student, Adminstudent)
admin.site.register(course, Admincourse)
```

→ Now run the commands; (all)

→ runserver — copy Ipaddress & past eit with /admin;

→ goto adminsite & login it.

→ you can see models on the admin home page, like below;

⟹

MtmApp	
courses	+add / change
students	+add / change

→ click on add form on student model & gives data;

⟹ Add student

sno	10	11	12
sname	sai	satya	nani
sloc	Hyd	Bang	pune

→ click on add form of course model & gives data;

⟹ Add courses

cno	100	101	102
cname	Python	Django	oracle
cfee	1500	2000	500

→ Django will create a new table to store the relationship of these parent & child tables.

→ goto mysqldb & check the tables.

⟹ select * from mtmapp_student;

⟹

id	sno	sname	sloc
1	10	sai	Hyd
2	11	satya	Bang
3	12	Nani	pune

⟹ select * from mtmapp_course;

⟹

id	cno	cname	cfee
1	100	Python	1500
2	101	Django	200

>>> select * from mtmapp_course_student;

=> id cno sname cname student_id

id	cid	student_id
1	1	1
2	1	2
3	3	3

—> Many to many not supports all cascading rules.
generally, if we delete a a parent record then automatically corresponding child record also will be deleted, when we set a CASCADE in a child table, but in this relationship the child record will not be deleted, beacuse, the same child record will be depended on multiple parents.

* for example; python course is learning by sai & satya if we delete a sai, the corresponding child, i.e. python course is not deleted, beacuse same python course is learning by other student satya also.

—> If we delete a sai & satya student, the child python course is not deleted.

—> If we delete a all student from parent table then all relationship from course_student table will be removed, but the courses from courses table will not be deleted, but course_student table will be empty.

* Model Inheritance : Django models supports different types of Inheritance, like any other language (like python, etc)..

→ It is nothing but the, deriving a fields, from one model to another.

→ It is also called as "model-Inheritance".

* Different types of model Inheritance : following are the different types of Inheritance ; (4)

① Abstract model Inheritance : Abstract model is a model, which has the common fields of other models.

→ If any fieldname is repeating in multiple models, then we can create a seperate models, then we can specify the common fields of all other models.

→ we have a set a "abstract = True", in a meta model of a new / other model [commonInfo], so that, this new model is called as "abstract-models".

→ If we want to derive the fields of abstract models to other models, then we should use class for all other models.

⟹ project name : Abstractpro

→ App name : Abstractapp.

→ db name : Abstractdb.

→ goto settings.py & configure the database & Installed apps;

→ goto models.py file & write below code;

⟹
```
class commoninfo(models.Model):
    name = models.charField(max_length=40)
    loc = models.charField(max_length=20)
    class Meta:()
        abstract = True

class student(commoninfo):
    sid = models.IntegerField()
    def __str__(self):
        return self.name
```

```
class Employee(commoninfo):
    sal= models.IntegerField()
    def __str__(self):
        return self.name
```

```
class customer(commoninfo):
    salesamt = models.IntegerField()
    def __str__(self):
        return self.name.
```

→ goto admin.py & write the below code;

```
=> from .models import student, Employee, customer,
                                    commoninfo

    class Adminstudent(admin.ModelAdmin):
        list_display ['name', 'loc', 'fee']

    class AdminEmployee(admin.ModelAdmin):
        list_display ['name', 'loc', 'sal']

    class Admincustomer(admin.ModelAdmin):
        list_display ['name', 'loc', 'salesamt']

admin.site.register(student, Adminstudent)
admin.site.register(Employee, AdminEmployee)
admin.site.register(customer, Admincustomer)
```

→ To run the commands; (all)

→ runserver. => copy the Ip address & paste on url along with /admin.

- NOTE: Django, is not creating a table for abstract, [commoninfo], beacuse it has no own data. ie. abstraction.

→ click on add form of student model, & give the proper data;

⇒ student

Name : [sai]

loc : [Hyd]

fee : [1500]

[save]

—> click on add form of employee table & give the data;

⇒ Employee

Name : [Ram]

loc : [Bang]

sal : [20,000]

[save]

—> click on add form of customer table & give the data;

⇒ customer

Name : [venkat]

loc : [pune]

salesamt : [40000]

[save]

—> goto database & check the database;

⇒>> select * from Abstractapp_student;

id	name	loc	fee
1	sai	Hyd	1500

⇒>>> select * from abstractapp_employee;

id	name	loc	sal
1	Ram	Bang	20,000

⇒>>> select * from abstractapp_customer;

id	name	loc	saleamt
1	venkat	pune	40,000

② Multilevel Model Inheritance : It is nothing but, deriving a fields from one model to other models, & again we can derive fields from this new model to another model, like (one after the other process).

While giving / inserting a data, we can go child model, we can give to all other model, through childmodel beacuse, the child model can have the fields of all parent models.

⟹ project name : multimodprd.

⟶ App name : multimodapp.

⟶ db name : multi6pmdb.

⟶ goto setting.py file & configure database & installed-apps

⟶ goto models.py file & write below code;

⟹
```
class Student(models.Model):
    sname = models.CharField(max_length=20)
    sfee = models.IntegerField()
    def __str__(self):
        return self.sname

class Employee(models.Model):          (student)
    ename = models.CharField(max_length=30)
    esal = models.IntegerField()
    def __str__(self):
        return self.ename.

class customer(models.Model):          (Employee)
    cname = models.CharField(max_length=40)
    csales = models.IntegerField()
    def __str__(self):
        return self.cname
```

⟶ goto admin.py & write a below code;

=> from .models import student, Employee, customer

```
class Adminstudent(admin.ModelAdmin):
    list_display ['sname', 'sfee']
class AdminEmployee(admin.ModelAdmin):
    list_display ['ename', 'esal']
class Admincustomer(admin.ModelAdmin):
    list_display ['cname', 'sales']
admin.site.register(Student, Adminstudent)
admin.site.register(Employee, AdminEmployee)
admin.site.register(customer, Admincustomer)
```

—> Run the commands; (all)

—> runserver. => copy the Ip address & paste on url, along with /admin.

—> student table contains, it's columns/fields like sname & sfee.

—> Employee table contains, it's column like ename & esal. and also Employee table contains, it's parent table (student) columns, like, sname, sfee, so, logically this table contains '4' columns.

—> customer table contains it's columns like, cname & csales, & also it got all columns from employee. i.e. (sname, sfee, ename, esal), so totally '6' columns/fields in the customer table.

—> now, we can goto customer table & we can give the data for all tables through customer table.

—> click on add form of customer & give the data beacuse, It has all fields of other tables also.

=> Add customer

sname: [saj]
sfee : [1500]
Ename: [satya]
esal : [40.000]
cname: [ram]

[SAVE]

—> If we click on save button then, the values of a sname & sfee columns, will goto student table & insert these.
—> The values of a ename & esal will goto employee table & insert in the table.
—> The values of cname & csales will insert into customer table.
—> now, goto database & check the data;

==>>> select * from multimodapp-student;

=>

id	sname	sfee
1	sai	1500

==>>> select * from multimodapp-employee;

=>

id	student-ptr-id	ename	esal
1		satya	40,000

==>>> select * from multimodapp-customer;

=>

employee-ptr-id	cname	csales
1	Ram	50,000

③ Multiple Model Inheritance : In this model Inheritance, a child model can inherit the fields from multiple parent models, into single child model, at one level.
—> The child model can have all fields of all parent models, so while giving data, we just goto child model, & give the data for all parent models.
—> we have to change the name of default auto increment fields of all parent models, beacuse all parent fields (including default id field), will be derived into child table.

So, multiple ids will be derived into child table, actually a table contains only one column with same name, if we take multiple columns with same name, then it will be error.

==> project name: multiplemodproo.
--> app name: multipleapp.
--> db name: multipleopmdb.
--> goto setting.py & configure database & Installed_apps;
--> goto models.py & write below code;

```
==> class Employee(models.Model):
        eid = models.AutoField(primary_key=True)
        eno = models.IntegerField()
        ename = models.charField(max_length=20)
        esal = models.IntegerField()
        def __str__(self):
            return self.ename
    class customer(models.Model):
        eid = models.AutoField(primary_key=True)
        cno = models.IntegerField()
        cname = models.charField(max_length=20)
        csales = models.IntegerField()
        def __str__(self):
            return self.cname

    class Student(Employee, customer):
        sno = models.IntegerField()
        sname = models.charField(max_length=40)
        sfee = models.IntegerField()
        def __str__(self):
            return self.sname
```

--> now, goto admin.py file & write below code;

```
==> from .models import Employee, customer, student
    class AdminEmployee(admin.ModelAdmin):
        list_display ['eno', 'ename', 'esal']

    class Admincustomer(admin.ModelAdmin):
        list_display ['cno', 'cname', 'csales']
    class Adminstudent(admin.ModelAdmin):
        list_display ['sno', 'sname', 'sfee']
```

admin.site.register(Employee, AdminEmployee)
admin.site.register(customer, Admincustomer)
admin.site.register(student, Adminstudent)

—> To run the commands ; (all)

—> runserver.

—> goto admin site & login with admin home page;

—> click on add form of student model & give the data.

==> Add student

Eno : [10]

Ename: [sai]

esal : [40000]

cno : [101]

cname: [satya]

csales: [50000]

sno : [1]

sname: [nani]

sfee : [1500] [SAVE]

—> if we click on save button then, eno, ename, & esal will goto automatically in employee table. & again into the customer table also.

—> The sno, sname, & sfee will goto current table.

—> NOW, goto database, & check the tables;

==>> select * from multiplemodapp_employee;

==>

eid	eno	ename	esal
1	1	sai	40000

==>> select * from multiplemodapp_customer;

==>

eid	cno	cname	csales
1	100	satya	50000

==>>> select * from multiplemodapp_student;

==>

employee_ptr_id	customer_ptr_id	sno	sname	sfee
1	1	1	Nani	1500

④ proxy model Inheritance: proxy model is a model which controlls a non-abstract models.

—> proxy model is used to manage the non-abstract model, to insert the data or to remove the data in the non-abstract model.

—> we have to set proxy = True, in meta class of the child model, so that model is called as "proxy-model".

—> To performing CRUD operations on non-abstract model, then we just goto proxy model, & then perform CRUD operations, internally all these CRUD operations will perform on "non_abstract" model.

==> project name : proxyprj

—>app name : proxyapp.

—> db name : proxydpmdb.

—> goto setting.py file & configure database & Installed-apps

—> goto models.py file & write below code;

```
==> class student(models.Model):
        sno=models.IntegerField()
        sname=models.CharField(max_length=40)
        sloc=models.CharField(max_length=60)
        def __str__(self):
            return self.sname.
    class studentproxy(student):
        class Meta:
            proxy=True.
```

—> goto admin.py file & write below code;

```
==> from .models import student, studentproxy
    class Adminstudent(admin.ModelAdmin):
        list_display ['sno', 'sname', 'sloc']
    class Adminstudentproxy(admin.ModelAdmin):
        list_display ['sno', 'sname', 'sloc']
admin.site.register(student, Adminstudent)
admin.site.register(studentproxy, Adminstudentproxy)
```

—> To Run the commands; (all)

—> runserver

—> copy the Ip address & login to admin site & perform the CRUD operations, on proxy model.

—> ① Inserting data ② updating ③ Deleting data.

* Create a Allrelationship models, in single project;

=> project name : Alltypesrelpro.

—> App name : Alltypesrelapp.

—> db name : Alltypesrelpmdb.

—> goto settings.py & configure database & Installed-apps;

—> goto models.py file & write below codes;

```
=> class publisher (models.Model)
        pname = models.charField (max_length=20)
        ploc = models.charField (max_length=30)
        pmail = models.EmailField (max_length=20)
        def __str__(self):
            return self.pname

    class Author (models.Model):
        aname = models.charField (max_length=20)
        aloc = models.charField (max_length=30)
        aincome = models.IntegerField()
        def __str__(self):
            return self.aname

    class student (models.Model):
        sname = models.charField (max_length=20)
        sloc = models.charField (max_length=30)
        sfee = models.Integerfield()
        def __str__(self):
            return self.sname

    class Book (models.Model):
        publisher = models.OneToOneField (publisher, on_delete=
                                          models.CASCADE)
        Author = models.ForeignKey (Author, on_delete=models.
                                    CASCADE)
```

```
    bname = models.CharField(max_length=20)
    bcost = models.IntegerField()
    def __str__(self):
        return self.bname
```

—> goto admin.py file & write below code;

```
=> from .models import publisher, Author, student, Book

    class Adminpublisher(admin.ModelAdmin):
        list_display ['pname', 'ploc', 'pmail']
    class AdminAuthor(admin.ModelAdmin):
        list_display ['aname', 'aloc', 'aincome']
    class Adminstudent(admin.ModelAdmin):
        list_display ['sname', 'sloc', 'sfee']
    class AdminBook(admin.ModelAdmin):
        list_display ['bname', 'bcost']
admin.site.register(publisher, Adminpublisher)
admin.site.register(Author, AdminAuthor)
admin.site.register(student, Adminstudent)
admin.site.register(Book, AdminBook)
```

—> To run the commands; (all)

—> runserver.

—> login to admin site & goto login home page.

—> click on add form on publisher model & give the data.

—> click on add form on Author model & give the data.

—> click on add form of student model & give proper data.

—> click on add form of BOOK & give a data;

—> NOW, goto database & check the data & tables;

```
=>>> select * from Alltypesrelapp_publisher;
=>>> select * from Alltypesrelapp_author;
=>>> select * from Alltypesrelapp_student;
=>>> select * from alltypesrelapp_book;
=>>> select * from alltypesrelapp_book_student;
```

* create a project to upload Images & Files :

=> If we want to upload images then, we should use ImageField.

—> If we want to upload files then, we use a FilesField

here, we create a one base folder with any name & we will create two subfolder in this base directory those are;

① files folder to store files.

② Images folder to store Images.

•NOTE: we don't create a folder manually, they will create automatically, when the run for first time.

=> project name: Filespro.

—> appname : filesapp.

—> goto setting.pj file, & take the following three lines after import-os line in setting.PJ;

=> BASE-DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))

MEDIA-URL = '/media/'

MEDIA-ROOT = os.path.join(BASE-DIR, 'media')

—> goto models.pj file & write below code;

=> class student(models.Model):

sno = models.IntegerField()

sname = models.charField(max_length=20)

stoc = models.charField(max_length=40)

Image = models.ImageField(upload_to="Images")

profile = models.FileField(upload_to="files")

def __str__(self):

return self.sname.

—> goto admin.pj file & write below code;

=> from .models import student

class Adminstudent(admin.ModelAdmin):

list_display ['sno', 'sname', 'stoc', 'image', 'profile']

```
admin.site.register(student, Adminstudent).
```

* Templates : A Template is nothing but the raw string, which contains one or more variable or tags.

→ Any [redacted] / text which is sorrounded by double curely braces, is called as variables.

=> {{ name }}
{{ company }}

→ A variable says "Inserting - something".
→ A text which is sorrounded by a curely braces & modulos symbol (·|·), that is called as "tag"
→ {·|· for ·|·}
{ ·|· name ·|· }
→ A tag says "do - something".
→ we will create a templates in a different ways;
→ we can create a template in a python shell also, & we can create dynamically in a html file, then we use file name.

① Template Creations : we have to import Template from django. Template.
→ we can use a multiple variables in a single template

* Through python shell : [python manage.py shell]

```
>>> from django.template import Template, context
>>>t= Template ("Hello, My name is {{ name }} and i
                learned {{ course }} and i also got job
                {{ com }}")
```

* context : A context is a dictionary which a "key-value" pairs.
→ The key names as must be same as a variables in a template.
→ we have to import context from "django.template" to create a context.

```
>>> from django.template import context
>>> c = context ( {'name': 'vikas', 'course': 'Django', 'comp': 'TCS'})
```

* render : It is used to give the mapping between context & Template.

—> template . render (context)

```
>>> x = t. render(c)
    print (x)
```

=> Hello, my name is vikas and i learned Django and i also got a job . TCS .

* creating a Template through the pycharm :

=> goto pycharm

—> got file —> New project.

—> select django, in the left side.

—> select proper interpreter.

• NOTE : If we create a project in pycharm then Django specifys appname & Installed app's automatically.

—> we have to configure the templates options in setting .py.

—> goto setting.py file;

```
=> TEMPLATES = [
        'BACKEND': 'django.template.backends.django.Django
                                                Template',
        'DIRS': [os.path.join (BASE_DIR, 'templates')],
        'APP-DIRS': True,
```

—> A 'BACKEND' is a dotted python path to the template engine.

—> A Django supports two types of templates those are ① Django Templates ② Jinja 2 .

—> but, the Default engines is DjangoTemplates.

* DIRS : It is a list of html / template paths.

→ If we create a project in pycharm then Django creates a templates folder bydefaults & also it gives the path to the templatefolder.

→ If we create a project through command prompt & open pycharm, then we have to give the path of a templates folders in 'DIRS' options.

=> dirs = [E:\\ tempfolder\\ temppro\\ templates', 'E:\\ tempfolder\\ temppro\\ templates']

* APP_DIRS : Bydefault app-dirs is set to True, that means the template engine will execute all installed apps defaults.

→ If we set false, then template engine will not execute default apps.

* create a template project for displaying a current "date-time";

→ goto views.py file & write below code;

```
=> from __future__ import unicode_literals
   from django.http import HttpResponse
   from django.template.loader import get_template
   import datetime as dt.
   def cur_datetime(request):
       now = dt.datetime.now()
       c = {'c_date': now}
       t = get_template("current_datetime.html")
       str = t.render(c)
       return HttpResponse(str)
```

=> create a html file with name current_datetime.html, & write below code;

⇒ <body>
<h1> welcome to Template world </h1>
 it is now {{ c_date }}
 Thanks for using template for example
</body>

→ goto urls.py file & write below code;

⇒ from tempapp import views
urlpatterns = [
url ('------'),
url ('time1', views.cur_datetime),
]

→ To run the server; copy the IP address, & paste on url path along with /time.

⇒

127.0.0.1:8000/time
Welcome to Template World it is now Aug. 26, 2018. 11:20 am "Thanks for using template for example"

* create a project to display details of Institute :

=> project name : Institute info

—> app name : Institute app

—> goto settings.py file & configure database, & Installed-apps ;

—> goto views.py file & write below code ;

=> from __future__ import unicode_literals
from django.shortcuts import render
from django.http import HttpResponse

def cur_date (request) :
    return render (request, "welcome.html")

—> create a template or html file with name as, welcome.html ; & write below code ;

=>
```
<body>
  <h1> Welcome Durgasoft </h1>
  <h2> New Batches in Durgasoft </h2>
  <p> <b> List of Databases </b> </p>

  <OL>
    <li> oracle </li>
    <li> mysql </li>
    <li> sql servers </li>
    <li> Nosql </li>
  </OL>
    <ul>
      <p> List of Languages </p>
      <li> python </li>
      <li> Django </li>
      <li> Java </li>
      <li> PHP </li>
    </ul>
    <h3> select required Framework </h3>
    <select>
    <option value="Django"> Django </option>
```

```
<option value="google App Engine"> google App Engine </option>
</select>
<hr>
  for more a Details
  <a href = "http://durgasoft.com/python-narayana-mvc">
    Click here </a>
</body>
```

—> goto urls.py file & write below code;

=> from app1 import views
(Institutecapp)

```
urlpatterns= [
    url [r'^$', views.cur_date),
    url [ r'^details/', views.cur_date),
]
```

—> run the server.

=> runserver. —> copy the ip address & paste on url along with /details.

* FORMS : A Form is a collection of "classes" or "inbuilt-library's".

→ forms are used to enteract with a websites.
→ creating a form is like a models.
→ we use a predefined class "forms.Form" to create a user defined forms.
→ we have to create a new python file "forms.py" in a applications & we can create a required forms in a forms.py file.
→ when we run a form then internally it convert in a html, then it will be created as form in the browser

* creating a forms :

→ create a python file with name forms.py in a application, & write below code;

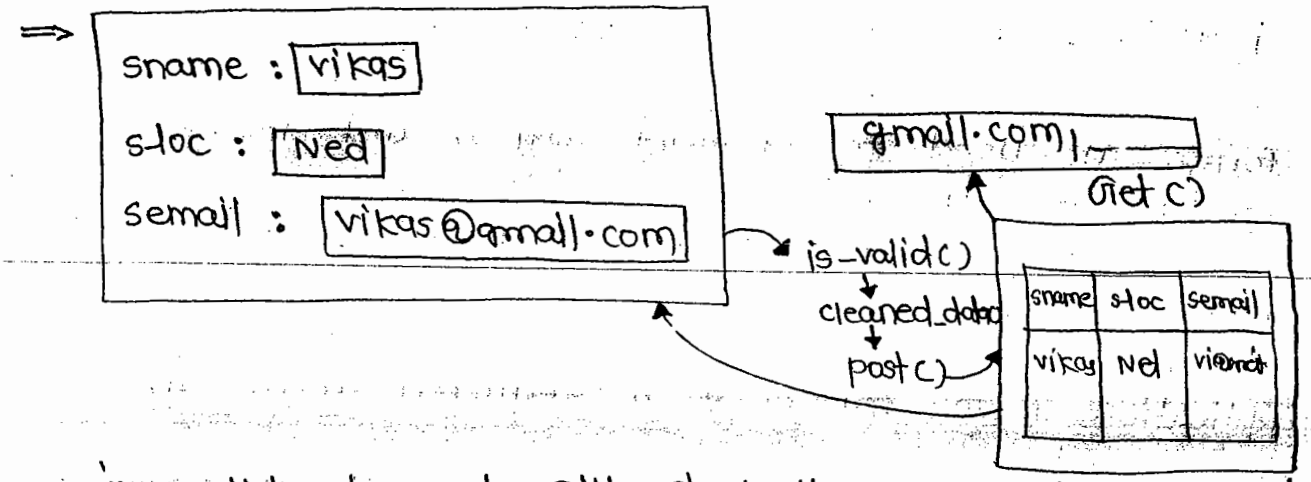
⇒ open forms.py file & write following code;

```
⇒ from django import forms
    class contactForm (forms.Form):
        sname = forms.CharField (max_length=20)
        sloc = forms.CharField (max_length=40)
        semail = forms.EmailField (max_length=40)
```

→ goto python shell, & run the following command;

```
⇒>> from simpformapp.forms import contactForm
>>> c= contactForm()
>>> print (c).
```

→ Now, it will displays a html code of contactForm.
→ Django, execute html code internally, & creates a form like below;



—> 'is_valid()' : It will check the form whether we have given all fields or not.

—> If we give the values for all required fields then 'IS_valid()', it will return True.

—> If 'IS_valid()' is True, then 'cleaned_data()' function it will convert data into dictionary format, like below.

=> {'sname': 'vikas', 'stoc': 'Ned', 'semail': 'vikas@gmail.com'}

—> 'post()' : it will take dictionary & post in the server then, the values will store in the database.

—> The server will return the response to the user by using 'post()' method, & also the server will change the url by using 'Get()' method.

* create a signup Form ;

=> create a forms.py file & write a below code.

```
from django import forms
```

=> class signupForm

```
    name = forms.charField (label='Name', max-length=20)
    location = forms.charField (label='Location', max-length=30)
    email = forms.EmailField (label='Email', max-length=40)
    salary = forms.IntegerField (label='salary')
```

→ goto views.py file & write below code ;

=>

```
from django.shortcuts import render
from .forms import signupForm.

def signupform(request):
    if request.method == "POST":
        form = SignupForm(request.POST)
    if form.is_valid():
        name = form.cleaned_data['name']
        location = form.cleaned_data['location']
        email = form.cleaned_data['email']
        salary = form.cleaned_data['salary']
        return render(request, 'result.html',
                      {'name': name,
                       'location': location,
                       'email': email,
                       'salary': salary})
    else:
        form = SignupForm()
        return render(request, 'signupform.html',
                      {'form': form});
```

→ create a html file with name as signupform.html, in template & write below code ;

=>

```
<body>
    <form action = "/signup/" method = "post">
        {% csrf_token %}
```

```
<input type="submit" value="submit" />
</form>
</body>
```

—> create a another one html file, with name "result.html" in templates folder, & write below code;

```
=> <h4> Thanks for signup </h4>
   <table>
     <tr>
       <td> Name: </td>
       <td> {{ name }} </td>
     </tr>
     <tr>
       <td> Loc: </td>
       <td> {{ location }} </td>
     </tr>
     <tr>
       <td> Email: </td>
       <td> {{email}} </td>
     </tr>
     <tr>
       <td> Salary: </td>
       <td> {{ salary }} </td>
     </tr>
     </table>
```

—> goto urls.py file & write below code;

```
=> from formapp import views as formapp_views
   urlpatterns = [
       url(r'^signup/', formapp_views.signupform),
       url('-----),
   ]
```

* creating a contact Form :

==> project name : contactformprp. [customer] [Employee]

—> App name : contactformapp.

—> db name : contactcrmdb.

—> goto settings.py, & configure Installed apps, also database ;

—> create forms.py file ; in applications & write below code;

```
==> from django import forms

    class ContactForm ( forms. Form):
        fname = forms. charField (label = "First Name",
            widget = Forms. TextInput (
                attrs = {
                    'class' : 'form-control',
                    'placeholder' : 'Enter your first name'
                }
            )
        )
        lname = forms. charField (label = "Last Name",
            widget = forms. ~~charfield~~ TextInput(
                attrs = {
                    'class' : 'form-control',
                    'placeholder': 'Enter your last name'
                }
            )
        )
        username = Forms. charField (label = "User Name",
            widget = forms. TextInput (
                attrs = {
                    'class' : 'form-control',
                    'placeholder' : 'Enter your user name'
                }
            )
        )
        password = forms. charField (label = "password"
            widget = forms. passwordInput(
                attrs = {
```

```
        'class' : 'form-control',
        'placeholder' : 'Enter your password'
      }
    )
  )
  Email = forms.EmailField(label="Email-ID",
      widget=forms.EmailInput(
          attrs={
            'class': 'form-control',
            'placeholder': 'Enter your email id'
          }
      )
  )
  mobile = forms.IntegerField(label="Mobile Number",
        widget = forms.NumberInput(
            attrs={
              'class' : 'form-control',
              'placeholder' : 'Enter your mobile number'
            }
        )
  )
```

—> goto views.py file & write below code;

```
=> from django.shortcuts import render.
   from django.http import HttpResponse
   from contactformapp import forms
   From contactformapp.forms import contactForm
   def index(request):
       if request.method == "POST":
           forms = contactForm(request.POST)
           if form.is_valid():
               return HttpResponse("data successfully
                                    Inserted")
           else:
               print(forms.errors)
       else:
           form = contactForm()
```

return render(request, 'indexcontact.html', {'forms': form3})

—> create a html file with name indexcontact.html file in template folder, & write below code;

```
<html>
<head>
<title> contact </title>
```

-5 lines

```
<style>
    .container {
        background-color : aqua;
    }
    form {
        bg-color : cadetblue;
        border : 2px solid red;
        border-radius : 10 px;
        padding : 5 px 28 px;
    }
    body {
        bg-color : red;
    }
</style>
<body>
    <div class = "container">
        <h2> contact us </h2>
            <div class = "col-md-4">
                <form method = "post">
                    {% csrf_token %}
                    {{ forms }} <br>
                    <center> <input type="submit" value=
                        "submit" class = "btn btn-default
                    btn-md"> </center>
```

```
    </form>
    </div>
    <div class="col-md-8">
        <h1><marquee behavior="alternate" "scrollamount=5">
            welcome to <b> Django Framework </b>

        <| marquee>
    </div>
  </div>
</body>
</html>
```

—> goto urls.py file & write a below code;

```
=> from django.conf.urls import url
   from django.contrib import admin
   from Contact formapp1 import views
   urlpatterns=[
          url('-----'),
          url( r'^$', views.index),
        ]
```

—> run the server.

—> python manage.py runserver. & copy the Ip address & paste in the browser, then it will display the contact form.

=>

Contact US

first name

Last name

username

password

Email ID

mobile no

Welcome to Django Framework.

→ After giving the data, In each field then, we click on submit button, then it displays Data successfully inserted.

→ This form will not save the data (user values) anywhere, beacuse we did not configure model.

* Using static files [Displays images in Form] :

⟹ we use a images, html files, css files, & Javascript file & so on, in the project.

→ These, all files are called as static files.

→ we take all this files into different folders, then we specify all these folders into static folder.

→ we should load a static folder into html folder.

⟹ {% load static %}

→ IF we dont load a static folder in html file, then we can not use a required static files.

⟹ project name : staticimagespro.

→ App name : staticimagesapp.

→ create a python file with name forms.py file in a application & write below code;

```
⟹ from django import forms
   class LoginForm ( forms.Form):
       username = forms.charField (label="User Name",
           widget = forms.TextInput (
               attrs = {
                   'class': 'form-control',
                   'placeholder': 'Enter your username'
               }
           )
       )
```

```
    password = forms.IntegerField(label="password",
            widget = forms.passwordInput(
                attrs = {
                    'class': 'form-control',
                    'placeholder': 'Enter your password'.
                }
            )
        )
```

—> goto views.pj file & write below code;

```
=> from django.shortcuts import render
   from django.http import HttpResponse
   from staticimagesapp import forms
   from staticImagesapp.forms import LoginForm
   def index(request):
       if request.method == "POST":
           form = LoginForm(request.POST)
           if form.is_valid():
           return render(request, 'output.html')
           else:
               print(forms.errors)
       else:
           form = LoginForm()
           return render(request, 'display.html',
                                   {'form': form})
```

—> create a static folder, which is parallel to the manage.pj.

—> create a images folder, which is sub folder in the static folder for store the images corresponding it.

—> create a html file, with name as, 'display.html', & write below code;

```
=> <html>
   <head>
   <title> Login Form </title>
```

5 line bootstrap

```
<style>
    h1 {
        bg-color : Blue;
        text align : center;
    }
    Form {
        border : solid ;
        padding : 50px 50px 50px 50px;
        border : 4 px solid red;
        bg-color : aqua;
        text-align : center;
        border-radius : 10px;
        margin-top : 300px;
    }
    img {
        min-width : 20 px;
        min-height : 70 px;
        max-width : 100.1. ;
        max-height : 200.1. ;
        position : relative;
    }
</style>
</head>
<body>
    <h1> <b> First Batch student </b> </h1>
    <div class ="container1">
      <div class = "row">
          < div class ="item">
      <center> <img src="{ .1. static "images/student1.jpg
                                .1. }" alt="students" </center>
      </div>
```

```
<div class="container2">
    <div class ="row">
      <div class = "col-md-offset-4 col-md-4 ">
       <form action = "/output.html" method="post">
                {% csrf_token %}
                {{ form3 }}
              <center> <input type ="submit" value='submit'
                 class="btn btn-default btn-md">
              </center>
       </form>
      </div>
    </div>
  </div>
</body>
</html>
```

→ create a another one html file, with name as a output.html & write below code;

```
<html>
<head>
<title> output Form </title>
```

5 img
Btsap

```
<style>
    img {
      position: relative ;
      margin center : 300px ;
<style>
</head>
  <body>
     <h2> <b> This is memorial Moments </b> </h2>
       <div class = "container 1">
```

```
        <div class ="tem">

    <center> img src= "{% static "images/motivational.jpg" %}"
                        alt= "students"
     </center>

   </div>
   </div>
   </div>
   [redacted]
   </body>
   </html>
```

—> goto urls.py file & write below code;

```
==> from staticimagesapp import views
      urlpatterns = [
            url ('-------'),
            url ('output$', views.index)
        ]
```

* Storing product data form in the database :

==> project name: productstorageproj.

—> App name : productstorageapp.

—> db name : productdb.

—> goto settings.py file & configure database & Installed apps;

==> goto models.py file & write below code;

```
==> class productData (models.Model):
          product_name = models.CharField (max_length=40)
          product_cost = models.FloatField ()
          product_color = models.CharField (max_length=20)
          product_desc = models.CharField (max_length=40)
```

→ goto admin.site & write below code;

⇒ from .models import productData

```
class AdminproductData (admin.ModelAdmin):
    list_display ['product_name', 'product_cost',
                  'product_color', 'product_desc']

admin.site.register (productData, AdminproductData)
```

→ goto forms.py file, in applications level & write below code;

⇒

```
from django import forms
from .models import productData

class productData(forms.Form):
    product_name = forms.CharField (label="Enter product
                                    name",
        widget= forms.TextInput (
            attrs={
                'class' : 'form-control',
                'placeholder' : 'Enter product name'
            }
        )
    )
    product_cost = forms.CharField (label="product cost",
        widget= forms.TextInput (
            attrs = {
                'class' : 'form-control',
                'placeholder' : 'Enter product cost'
            }
        )
    )
    product_color = forms.CharField (label="product color",
        widget = forms.TextInput (
            attrs={
                'class' : 'form-control',
                'placeholder' : 'Enter product color'
```

```
        )
    )
    product_desc = forms.CharField(label="product Desc",
            widget = forms.TextInput(
                    attrs={
                        'class' : 'form-control',
                        'placeholder' : 'product desc'
                                                'Enter
                    }
            )
        )
```

→ goto views.py file & write below code;

```
=> from django.shortcuts import render
   from .models import productData
   from .forms import productForm
   from django.http import HttpResponse.
   def productdetails(request):
       if request.method=="post":
           form=productForm(request.post)
           if form.is_valid():
               pname= request.post.get('product_name')
               pcost = request.post.get('product_cost')
               pcolor = request.post.get('product_color')
               pdesc = request.post.get('product_desc')
               data= productData(product_name=pname,
                                 product_cost = pcost,
                                 product_color = pcolor,
                                 product_desc = pdesc)
               data.save()
               data = '<h1> insertion is successfully done
                                                    </h1>'
               return HttpResponse(data)
           else:
               print(form.errors)
       else:
```

return render(request, 'details.html', {'form': form})

—> create a html file, with name details.html & write code;

=> <html>
<head>
<title> Hello Django </title>

<style>
body {
bg-color : cadetblue;
}
.container {
bg-color : yellowgreen;
}
form {
border : 5px solid red;
border-radius : 5px;
padding : 15px 15px 15px 15px;
bg-color : aquamarine;
}
h3 {
text-align : center;
color : red;
}
</style>
</head>
<body>

<h3> Enter product Details here</h3>
<form method="post">
{% csrf_token %}

```
        {{ form }}
    <center> <input type ="submit", value="submit",
                class ="btn btn-default btn-md">

    </center>
    </form>
  </div>
 </div>
</body>
</html>
```

—> goto urls.py file & write below code;

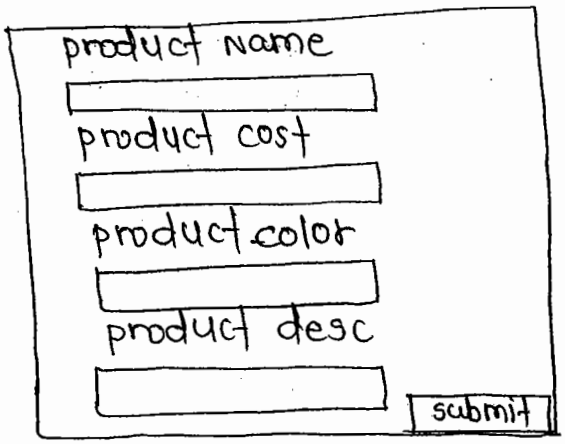
```
=> from productstorageapp import views
   urlpatterns = [
        url(r'^admin/', ----),
        url(r'^$', views.productdetails),
    ]
```

—> To run the commands; (all)

—> runserver.

—> click on ip address & copy, & paste on url path with empty.

=> Enter product details



—> Insertion is successfull done, after we click on, message will display here.

—> goto database & check the data;

=> select * from productstorageapp_productdata;

* Admin site : It is used to manage the website or management of website.

—> we have to create username & password for admin site & give to the admin people of website., so that they can login to admin site & manage the website & modifying content of website, like they can add new data or replace the data or delete the data from the website.

—> if you don't create username, & password then data of website, can not be modified.

—> we use createsuperuser command to create a username & password.

—> once, if we login to the admin site, then we have some permissions.

All. permissions, are controlled by main user i.e. (superuser).

—> superuser can create a other users also, (new user/ other user).

—> superuser can give required permission to the new user.

* Available permissions in Django :

① Active : It is a default permissions for every new user it means, this is user of active member of current website

—> If we want to delete a any existing user from the admin site, they simply we can inactive this user.

② Staff status : If we specify this permission to any user, then this user can login to admin site, with this his username & password.

—> If we want to assigne this permission, to a specific user, then he must be active member of this in the website.

—> This user doesn't have any permissions bydefault.

—> A main user can assigne the required permission to this user.

③ superuser status : If we assigne this permission to any user he can have all permissions like superuser but, he can not login to admin site, with his username & password.

—> If we / he wants to login to website, admin site he must staff status along with superuser.

—> If we assigne both staff status & superuser status to a specific user, then he has all permissions like main user.

* Managing cookies : A HTTP is a stateless protocols, that means, when we request to server, then it has no ideq new all requesting a same person to or same page for first time.

or, you are same person to has visited same page, multiple time before itself.

—> This lack of stateless was a big problems along q developers; To avoide, this problems, then cookies was invented.

* What is cookies : A cookie is a small piece of data, that is stored in q user's browser, which is sent by the server.

* How cookie works :

① The browser sends the request to the server.

② The server sends the response along with one or more cookies to the browser.

③ The Browser saves the cookies, which is received from the server.

④ From now onwards, the browser will send the cookies, to the server, it will send q cookies to server along with q each request until the cookies expires.

⑤ when cookie expired, it is removed from the browser.

* How to create a cookies [program]:

⟹ goto views.py & write below code;

⟹ from django.shortcuts import render
from django.http import HttpResponse

```
def index(request):
    return HttpResponse("<h4> cookie was first by a
                         programmer"
                        "named Louis Montulli in 1994 <br>"
                        "at Netscape communications"
                        "in their Netscape Browser. </h4>")

def test_cookie(request):
    if not request.COOKIES.get('color'):
        response = HttpResponse("cookie set")
        response.set_cookie('color', 'blue')
        return response
    else:
        return HttpResponse("your favourite color is (0)".
                            format(request.cookies['color']))
def count_cookie(request):
    if not request.COOKIES.get('visits'):
        response = HttpResponse("This is your (0) visit".
        response.set_cookie('visits', str(visits)) format(visits))
    return response
def delete_cookie(request):
    if request.COOKIES.get('visits'):
        response = HttpResponse("cookies cleared")
        response.delete_cookie("visits")
    else:
        response= HttpResponse("we are not tracking you").
    return response.
```

⟶ goto urls.py file & write below code;

⟹
```
from django.contrib import admin
from django.urls import path
from cookiesapp import views
urlpatterns = [
    path('------'),
    path('founder', views.index),
    path('count', views.count_cookie),
    path('delete', views.delete_cookie),
]
```

* Sessions : we know that cookie allows us to a store the data in a browser easily, even though cookies are usefull, they have following problems;

① An attacker can modify the content of a cookie, that could break our applications.
② we can not store, sensitive data.
③ we can only store a limited amount of data in cookies.
④ Most browser's don't allow a cokies to store more than 4 kb of data.
⑤ Breaking data into multiple cookies can cause to much overhead in each request.
⑥ Further, we can not even depend on a number of cookies allowed by the browser.
⑦ To overcome this problems we use a sessions.

* ① when we use sessions the data is not stored directly in the browser, it will store in the server.
② Django creates a unique random string called session ID. or SID, & Django associates the SID with data.
③ The server send's the cookie's named session ID or SID as a value to the browser.
④ on the requesting web page, the browser sends, a request along with SID to the server.
⑤ Django then uses this SID to retrieve the data & makes to the accessible in your code.
⑥ SID is a generated by django & it is 32 characters long random string.

so, it is almost impossible to guess by any hackers.

—> project name : sessionpro.

—> App name : sessionapp.

—> goto views.py & write below code;

```
=> def visitcount(request):
        num_visit = request.session.get('count',0)
        num_visit = num_visit +1.
        request.session['count']= num_visit
            return render(request, 'show.html',
                    {'num_visits': num_visit})
```

—> create a html file as 'show.html' in templates;

```
=> <h1> welcome to session management </h1>
   <h2> you have visited the site for <b>
            {{ num_visits }} </b> time(s)
   </h2>
```

—> goto urls.py file & write a below code;

```
=> from sessionapp import views
    urlpatterns = [
        url ('------')
        url (r'^visit1', views.visitcount),
    ]
```

—> run the server & copy the Ip address, along with ^visit1.

=>

https://127.0.0.1:8000/^visit/
Welcome to session management you have visited the site for 1 time(s)

* Class based views : Actually views are two types;

① Function based views ② class based views ;

① we create a function based use 'def' python keyword, but class based views are created by using 'class' python keyword.

② In function based views, we write complete code, manually, but in class based views, we use existing class name as based classes.

③ function based views takes more time, but class based views takes less time.

④ class based views are used to create end point url in Rest-API, compare to django views.

—> project name : Entryclassbasedpro [family members ME)

—> app name : Entryclassbasedapp

—> db name : genericdb.

—> goto settings.py file & write apps name & also configure a database ;

=> goto models.py & write below code;

```
—> class Member(models.Model):
        name = models.CharField(max_length=20)
        desc = models.CharField(max_length=40)
        def __str__(self):
            return self.name
```

—> from .models import Member

admin.site.register(Member)

—> create forms.py file & write below code;

```
=> from django import forms
   class FamilyMemberForm(forms.Form):
       name = forms.CharField(
           widget = forms.TextInput(
```

```
            attrs = {
                'class' : 'form-control',
                'placeholder' : 'Enter your Name'
            }
        )
    )
    desc = forms.CharField(help_text="Write a brief about
                                        person",
        widget = forms.TextInput(
            attrs = {
                'class' : 'form-control',
                'placeholder' : 'Enter your desc'
            }
        )
    )
```

→ goto views.py & write below code;

```
⇒ from django.shortcuts import render
  from django.views import generic
  from classbasedview app.forms import FamilyMemberForm
  from .models import Member.
  def makeentry (request):
      if request.method == "POST":
          form = FamilyMemberForm (request.POST)
          if form.is_valid():
              name = request.post.get('name', '')
              desc = request.post.get('desc', '')
              member = Member (name=name, desc=desc)
              member.save()
              form = FamilyMemberForm()
              return render (request, 'genericviews/makeentry.
                                  html', {'form' : form})

      else:
          form = FamilyMemberForm()
          return render (request, 'genericviews/makeentry.
                                  html', {'form' : form})
```

```
class Indexview (generic.listview):
        context_object_name = 'list'
        template_name = 'genericviews/index.html'
        def get_queryset (self):
            return Member.objects.all()
```

```
class Detailsview (generic.Detailview):
        model = Member
        template_name = 'genericviews/detail.html'
```

—> create a html file with name makeentry.html, in genericviews folder, which is created in template folder;

==> <html>

<head>

5 lines bistro

```
<style>
    body {
        bg-color : blue;
    }
    .container {
        bg-color : darkseagreen;
        border-radius : 10 px;
        margin-top : 50 px;
        margin-right : 100 px;
        margin-left : 100 px;
    }
    Form {
        border : 5 px;
        border-radius : 5 px;
        border-color : black;
        background-color : coral;
        margin-bottom : 10 px;
    }
```

```
<body>
<div class ="container">
    <h1 class= "text center"> please make Family-member
                                    Entry's here </h1>
    <div class ="row">
        <div class ="col-md-4">
            <form action =" {% url 'genericviews : makeentry'
                        %} " method= "post" >
            { % csrf_token % }
            {{ % form }} <br> <br>
        <center> <input class= "btn. btn-primary" type ="submit"
                        value ="submit" > </center> <br>
            </form>
    </div>
</div>
</body>
</html>
```

—> create index.html file, in genericviews folder, & write below code;

```
=> {% if list %}
  <h1> member Details !</h1>
  <ul>
      {% for member in list %}
      <li>
        <a href="{% url ' genericviews : detail' member.id %}">
                        {{ member.name }} </a>
      </li>
        {% end for %}
      </ul>
        {% else }
    <h3> No Member Found </h3>
    {% endif %}
```

—> create a html file with name as detail.html in generic views folder, & write code;

=> <h1> {{ member.name }} </h1>
<h3> {{ member.desc }} </h3>

—> goto urls.py file in application level, & write below code;

=> from django.conf.urls import url
from classbasedviewsapp import views
app_name = 'generichviews'
urlpatterns = [
url(r'^$', views.IndexView.as_view(), name='index'),
url(r'^(?P<pk>[0-9]+)/$', views.DetailsView.as_view(),
name='detail'),
url(r'^makeentry$', views.makeentry, name='makeentry'),
]

—> goto urls.py file in project level & write below code;

=> from django.contrib import admin
from django.urls import path, include
urlpatterns = [
path('admin/', admin.site.urls),
path('genericviews/', include('classbasedviewsapp.urls')),
]

* Template Inheritance : It is the process of inheriting or deriving a html code of one file to another html file, like python class inheritance.

—> generally, we use a either extends or include to inherit the html file, if we use extends, then the current html file, will goto html file & executes there.

If we use include, then html code of base file will come to the current file & executes here.

generally, in all projects we use template inheritance, to reduce the length of code in the file.

—> project name: Temp Inheritance.

—> app name : blog_app.

—> goto views.py file & write below code;

```
=> from django import template
   from django.shortcuts import render, render_to_response
   from django.http import HttpResponse
   import datetime
   def today_is(request):
       now = datetime.datetime.now()
       return render(request, 'blog/datetime.html',
                                  {'now': now})
```

—> create a nav.html file in blog folder in a template & write a below code;

```
=> <nav>
       <a href="#"> Home </a>
       <a href="#"> blog </a>
       <a href="#"> contact </a>
       <a href="#"> career </a>
       {% block content %}
       {% endblock %}
   </nav>
```

→ create a datetime.html file in blog & write code;

=> <body>

{% extends 'blog/nav.html' %}

{% block content %}

{% if now %}

 current datetime is : {{ now }}

{% else %}

 now ~~evalt~~ variable is not available

{% endif %}

{% endblock %}

</body>

→ create urls.py file in application & write below code;

=> from django.conf.urls import url

from .import views

urlpatterns= [

url (r'^$', views.today_is),

]

→ goto urls.py file in project & write below code;

=> from django.contrib import admin

from django.conf.urls import url, include

from blog_app import views

urlpatterns = [

url('----'),

url (r '^$', include ('blog_app.urls')),

]

→ If we create a project in pycharm then, Django will map the default templates folder in template options in setting.py file.

→ If we create the project through shell & open's in

manually in setting.py file.

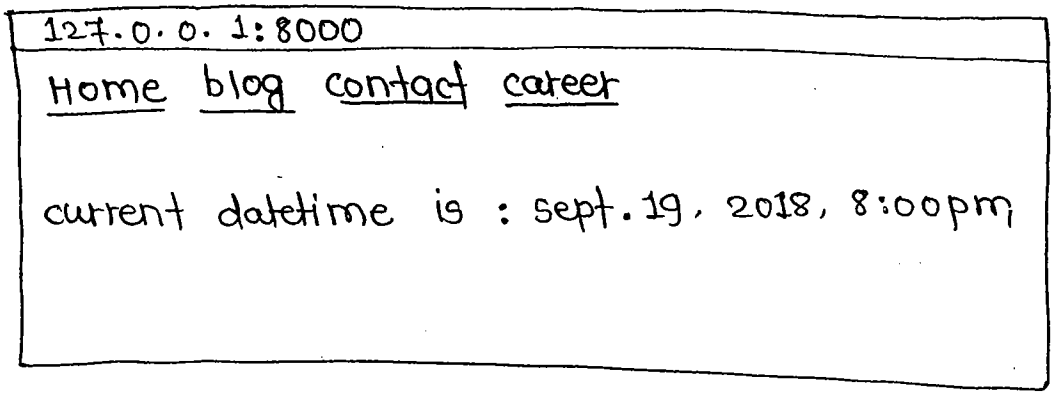
—> To run the commands.

=> python managin.py file

—> If we create the project through shell & opens in pycharm;

—> runserver, & copy ip address paste in the browser;

=>



* Cache management : Cache memory is used to save the dynamic web page.

—> when we request first time then the server will takes the request & process the operations, while sending a response, it will store the result of webpage in cache memory.

—> Django support different backends to store the cache file in different locations.

—> we can get the backends in the following path;

=> python 36-32|lib|site-packages|django|core|cache|backends.

—> so, following are the some backends of cache;

① memcached : If we want to save the cache file / cache data in remote server, then we use memcached.

—> goto setting.py file & configure memcached;

=>
```
CACHES={
    'default':{
    'BACKEND':'django.core.cache.backends.memcached
                .MemcachedCache'
```

```
    'LOCATION': '127.0.0.1:8001',
  }
 }
```

② Locmem: If we want to use for save the cache data in local memory / local server, then we use a locmem.

```
=> CACHES = {
      'default': {
          'BACKENDS': 'django.core.cache.backends.
                       locmem.LocMemcache',
          'LOCATION': '127.0.0.1:8000'
      }
  }
```

③ filebased: If we want to save the cache data any required directory, in local computer then we use a filebased.

```
=> CACHES = {
      'default': {
          'BACKEND': 'django.core.cache.backends.filebased.
                      FileBasedCache',
          'LOCATION': 'd:/folder2'
      }
  }
```

④ db: If we want to save the cache data in a database table, then we create a cache table by using createcache table command, then configure cache table.

```
=> python manage.py createcachetable mytab3
=> CACHES = {
      'default': {
          'BACKEND': 'django.core.cache.backends.db.
                      BaseDatabaseCache',
          'LOCATION': 'mytab3'
      }
  }
```

* Deplojement : It is nothing, but the deploying a django project into production server.

before, deploying we will perform some operations.

① we set "DEBUG=False", in setting file beacuse, if it is True, then django will goto each & every application of file, & debug the all file, which takes more time when we have multiple applications in the same project, & also it will create some temp files to store debugging data for every request to the server.

② when we run the project multiple time then it will create multiple temp files, if it creates multiple temp files then, it will become burdon on the server.

—> we will take all our modules, which are running in our machine into a new requirements file by wing "freeze" commands, then we run that requirements file in production server, so that all the modules which are running in our server, also will install in production server.

—> we share our project into production server then run the project in that machine.

—> if the project is running well then deplojement is completed from our side, later the client will by servers & domains will map those it.

* Form validations: To perform validation on the form to take a proper data & also to avoide a inproper data, like emailid must contains @gmail.com like;

—> mobile number field must allow 10 digits only.

—> both password & conferm password must be same.

—> we perform the validation in different levels.

① Field level validations ② Form level validations.

① Field Level validations : If we want perform validations for every field individually then, we go for field level validations.

here, we use a clean_(method / fieldname)

like ==> clean_username()

② form level validation : when we want perform the validation the comparing two or more fields then, we go for form level validations.

—> In form level validations we use clean() method, in this method, we perform the validations.

always to perform the validations in "forms.py" file only.

=> project name: validateprn.

—> app name : validateapp.

—> db name : validatedb.

—> goto settings.py file & configure database.

—> goto models.py file & write the below code;

```
=> from django.db import models
   class Reg(models.Model):
       username = models.charField(max_length=20)
       email = models.EmailField(max_length=10)
       password = models.charField(max_length=20)
       password2 = models.charField(max_length=20)
```

—> create a forms.py file in application's, & write below code;

```
=> from django import forms
   from django.contrib.auth import get_user_model
   User = get_user_model()
   class RegForm(forms.Form):
       username = forms.CharField(
           widget = forms.TextInput(
               attrs={
                   'class': 'form-control',
                   'placeholder': 'Enter user name'
               }
           )
       )
       email = forms.EmailField(
           widget = forms.EmailInput(
               attrs={
                   'class': 'form-control',
```

```
            'placeholder': 'Enter your email'
        }
    )
    password = forms.TextInput(
        widget = forms.passwordInput(
            attrs = {
                'class': 'form-control',
                'placeholder': 'Enter your password'
            }
        )
    )
    password2 = forms.TextInput( label='confirm password',
        widget = forms.passwordInput(
            attrs = {
                'class': 'form-control',
                'placeholder': 'reEnter your password'
            }
        )
    )

    def clean_username (self):
        username = self.cleaned_data.get ('username')
        qs = User.objects.filter(username=username)
        if qs.exists():
            raise forms.validationError("username
                                is taken already.")

        return username
    def clean_email (self):
        email = self.cleaned_data.get ('email')
        qs = User.objects.filter (email=email)
        if qs.exists() :
            raise forms.validationError ("Email
                                is already taken".)

        elif not 'gmail.com' in email :
            raise forms.validationError("email
                                has to end with gmail.com")

        return email
```

```
def clean(self):
        data = self.cleaned_data
        password = self.cleaned_data.get('password')
        password2 = self.cleaned_data.get('password2')
        if password2 != password:
                raise forms.validationError("password
                                must be same, like above")
        elif len(password) <=3 or len(password)>=7:
                raise forms.validationError("password
                                must be, atleast 8 chars")
        return data
```

→ goto views.py file, & write below code;

```
⇒ from django.contrib.auth import get_user_model
from django.shortcuts import render
from .forms import RegForm
from django.http import HttpResponse, HttpResponse
                                        Redirect
from .models import Reg
User = get_user_model()
def RegForm_page(request):
        form = RegForm(request.POST or None)
        context = {
                'form': form
        }
        if form.is_valid():
                print(form.cleaned_data)
                username = form.cleaned_data.get('username')
                email = form.cleaned_data.get('email')
                password = form.cleaned_data.get('password')
                new_user = User.objects.create_user
                                (username, email, password)
                print(new_user)
        return render(request, 'feedback.html', context)
```

—> create html file, with name 'feedback.html', & write below code;

==> <html>

<head>

5 lines btstrap [ ]

<style>

.container{

bg-color : cyan;

}

form {

bg-color : chocolate;

padding : 10px;

border-radius : 5px;

margin-top : 20px;

border : red 5px solid;

}

h1 {

font-family : cursive;

color : red;

</style>

</head>

<body>

 <h1> please Register your details</h1>

<form action=" " method="post">

{% csrf_token %}

{{ form}}

 <input class="text-center" type="submit"

```
    </form>
        </div>
    </div>
    </div>
    </body>
```

—> goto urls.py file & write below code;

```
=> from django.contrib import admin
   from django.conf.urls import url
   from validatcapp.views import Regform_page
   urlpatterns=[
        url('^admin/', admin.site.urls),
        url(r'^reg/$', Regform_page),
```

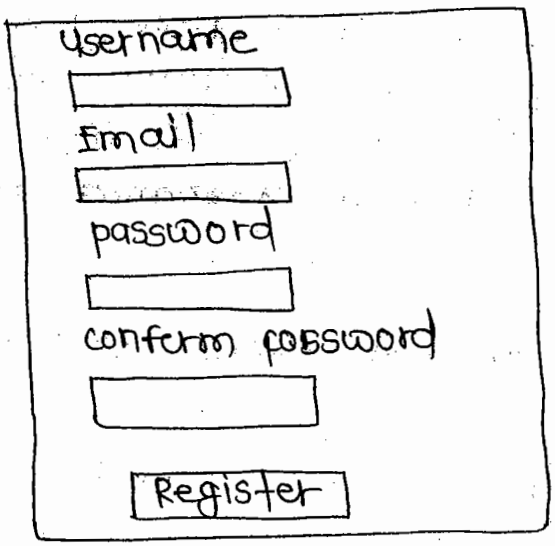
—> To run the commands ; (all)

—> runserver, [2020] port number [python manage.py runserver 2020]

—> copy address & paste on browser along with /reg.

—> A output will display like below;

=> please Register your details



—> now, we can perform the specified validation in each field.

* User Interface : [Authentication] :

—> we can create a new users & change the password from without admin sites also.

—> Django provides some default functions to create & change the username & password.

—> we can perform these operations by using these default model 'User' that means we import this user before performing the operations.

—> The current projects has user 'nani' & password '1234abcd' in a validatepro project.

Now, we change the password & also we create a new users in the same project.

* changing a password :

=> we can change a password to corresponding user through python shell, like below;

```
>>> from django.contib.auth.models import User
>>> user=User.objects.get(username='nani')
>>> user.set_password(123abc)
>>> user.save()
```

• NOTE : Now goto admin site & logout from nani user who has a password 'abcd1234" again with same user & by using 'abcd1234' then it will throw error, beacuse, we changed password from '1234abcd', to '123abc', so we should use a 123abc, for 'nani' user.

* creating a new user : [Through python shell]

```
>>> from django.contib.auth.models import User
>>> user=User.object.create_user(username='venkat', password=
                                 'venkat123', email='venkat@gmail.com')
>>> user.save()
>>> user.is_staff=True    # Giving staff_status permissions.
>>> user.save()
>>> user.is_superuser=True # Giving superuser permissions.
>>> user.save()
```

-> every user supports some default variables (field/ methods).

-> we can change a user_name like below;

```
>>> user=User.objects.get(username='vikas') # getting existing user
>>> user.username                          # displaying existing user
    'vikas'

>>> user.username = 'python vikas'  # changing username.
>>> user.save()
```

-> Now, we goto admin site & logout from 'vikas' user again try to login user with same, it will throw error. because username changed, to python vikas, Now we can login with same old password.

-> we can give users first_name & last_name through python shell also like below;

```
=> print(user) -> python vikas  # checking current user.
>>> user.first_name = 'python'  # giving first name.
>>> user.save()
>>> user.last_name = 'vikas'    # giving last name.
>>> user.save()
```

-> we can retriere all the data of a specific user like below;

```
>>> user.username      'python vikas'
>>> user.first_name    'python'
>>> user.last_name     'vikas'
>>> user.email         'vi@gmail.com'
>>> user.is_staff      True.
>>> user.is_active     True.
>>> user.last_login    datetime.datetime(2018, 9, 21 13, 1, 20)
>>> user.password      # hash coding.
```

generally, the password we convert as a Hash-coding that means we can not password as raw data (as it is).

→ If we want to see password raw data then we user create() to create user.

→ Django supports both create_user() & create()

→ If we create_user() then automatically password will convert as hashcoding.

→ If we use create() then password will not convert as has coding.

→ When we run ocreatesuperuser command then we don't use these two functions directly to create user, but here also the password will convert as hash coding, that means the default method is create_user() to create user

→ Now, create user by using create() function;

```
>>> u1 = User.objects.create(username='vikas', password='vikas@gmail',
        email='vikas@gmail.com')
>>> u2 = User.objects.get(username='vikas')
>>> u2.username        'vikas'
>>> u2.first_name      ''
>>> u2.last_name       ''
>>> u2.email           'vikas@gmail.com'
>>> u2.password        'vikas1234'        # password as raw code.
```

* Customizing admin site: we can customize the admin site by adding extra features to existing a admin page.

(1) list_display: It is used to display list of fields in the admin site.

(2) search_fields: It is used to provide a search box for a specific or n-number of fields, so that we can search the data of correspondlng fields in the search box.

③ list_filter : It is used to create a filter by using the unique values of the specified field, if we click or any specific unique value then it displays all the data which belongs to unique value specified it.

④ list_display_links : It is used to provide a link to the first field of list display by default. if we ~~want~~ not provide a link to any other field then we specify that field.
→ This link is wefull to update the data of a specific record.

⑤ list_editable : It is used to edit a specific field in the table itself.

```
=> class Adminstudent(admin.ModelAdmin):
       list_display = ['sid', 'sname', 'loc']
       search_fields = ['loc', 'sname']
       list_display_links= ['loc']
       list_editable = ['sname']
       list_filter = ['loc']
       ordering = ['-sname'] or ['sname'] # displays
                          Assending or desending orders.
   admin.site.register(student, Adminstudent)
```

9766250967

V

① what is Rest-API ?

② what is a web-services ?

③ what are the types of views ? which kind of views you used in your project ?

④ we have a table with 4 columns(id, name, sal, comm) & the table contains 5 rows. How to convert into JSON format.

⑤ what are the types of viewsets ?

⑥ what is the serialization & deserialization ?

⑦ what are different status codes that you used in your last project ?

⑧ what is the Architecture of your project. ?

⑨ what is team size of your project ?

⑩ can you alone handle the entire project ? [yes, i can]

⇒ Rest is a Representational state transfer, & it is Architectural style, which allows system to take a wide range of data & it gives the data to the client. any time in required format.

⇒ API it's stand for Application programming interface, which is an intermediate between two application, which communicate each other.

⇒ Web services, is a Restful web API, which is implemented by using Http methods (Get, post, put, & Delete) & it's principles.

⇒ Types of views: ① class based views ② Function based views. but, mostly we use a class based views, beacuse of code reusability & we can dicrease the size of our coding, in project.

⇒ There are two types of viewsets : ① APIviews ② Viewsets (modelviewset)

⇒ Serialization means, the way of model instance or querysets into native datatypes, then converted into JSON formats.

⇒ Deserializations, means, which is reverse process, which is takes the data from browser & then converts into JSON & convert into model instances.

⇒

* Introduction of JSON: JSON is also stands for Javascript object Notation.

→ JSON is a simple data exchange format, which helps to communicate between Javascript & serversite technology.

→ It starts with curely bracket {, } and also end it.

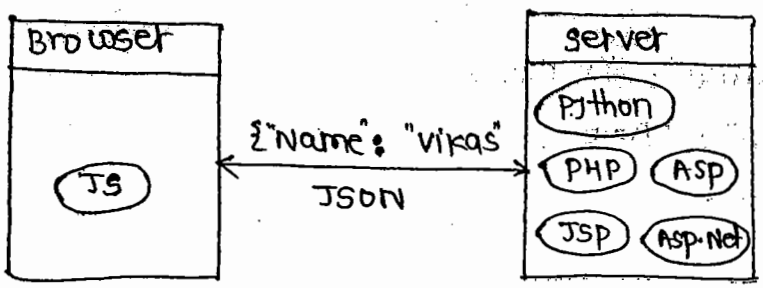
=> <schipt>

```
var x = { "custId" : 10, "custName" : "vikas" }
alert (x. custId);
```

</schipt>

→ JSON is a syntax for storing & exchanging data.

=>



→ It is lightweight data-interchange format, & based on subset of Javascript.

→ It can be used with most modern Language.

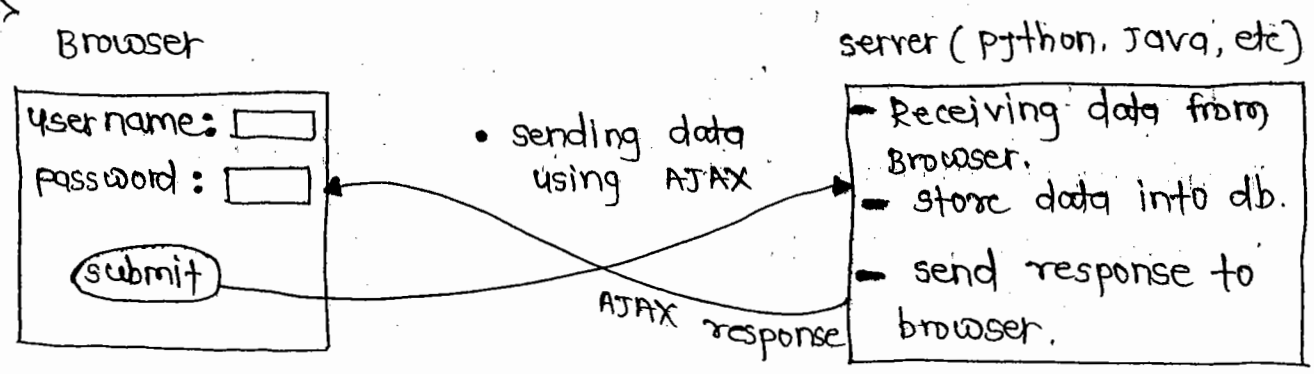
→ JSON is Language independent.

→ JSON is self-describing & easy to understand.

→ The JSON syntax is derived from Javascript, object notations syntax, but the JSON format is text only, code for reading & generating.

→ JSON data can be written in any programing Language.

=>



→ It is nothing but the concept of sending request to server & getting response from server into the same browser page, without refreshing the same page.

→ JSON must be an AJAX.

→ The Response will be sent from server to browser, in JSON format.

→ It is not an object, it is a "plain text", that looks like object.

→ JSON values cannot be one of the following datatypes;

① a function.
② a date.
③ undefined.

⇒
```
<script>
    var a = { "first name": "vikas",
              "Lastname": "Asle",
              "age": 24,
              "profession": "web developer"
              }
    document.write(a);
    // document.write(a.first name, a.Lastname, a.age);
    </script>
```

→ In the above JSON format, it is a valide or not, it can be checked through one website,

→ goto "JSONLint.com", copy the JSON format & check it.

* Datatypes :

① Numbers : (Int & float).
② String : (use double quotes only)
③ Boolean : (True/ false).
④ Array : (ordered list '0' or more values)
⑤ object : (unordered collection of key/value pairs)
⑥ NULL : (empty value).

* JSON syntax rules:

—> use key/values pairs { "Name" : "vikas" }.

—> use double quotes around key & values.

—> must use the specified datatypes.

—> file type is ".JSON"

—> MIME type is "Application/json".

```
==> <script>
    var x = {
        "firstname": "vikas",
        "Lastname": " Asle",
        "age" : 24
        }
    // document.write(x.firstname);
    // for (showalldata in x)
// modifying value;
// x.firstname="viraj";
// adding a new key value pair;
// x.middlename = "uttamrao";
// deleting a key value pair;
// delete x.middlename;
// for ( showalldata in x)
// {
// document.write(x [showalldata] + "<br>");
// }
var x= {
    "array": ["vikas", "uttamrao", "Asle"]
}
document.write (x.array);
var x = {
    "FIFA" : [
        {
        "countryname" : "Brazil",
        "Bestplayer" : "Neymar"
        },
```

```
{
"countryname": "portugal",
"Bestplayer" : "Ronaldo"
},
{
"countryname": "Argentina",
"Bestplayer" : "Messi"
}]
}
// document.write(x.FIFA[0].country / Bestplayer);
for (y in x.FIFA)
{
// document.write(y);
for( inner_y in x.FIFA[y]){
// document.write(x.FIFA[y][inner_y] +"<br>");
// }
// }
var z= {"name":"vikas", "Lastname" : "Asle" };
var data = JSON.stringify(senddata);
var m = JSON.parse(data);
document.write(m.name);
</script>
```

REST API

* Introduction of API:

⟹ API is stands for Application programming Interface.

⟶ An API is a intermediate software, which allows two Applications to talk each other.

* For example; if we click on any application on our mobile then first the application will check the for Internet then it will goto corresponding server with request from the user.

⟶ The server takes the data from the application & it will perform perform operation on the request & sends request to the application, which is on our mobile screen, & it will display to the user in specified manner.

* for Example; if we goto any Resturant then we takes a orders & give items as per orders.

so, here waiter will works as API, who takes orders from the customer & then brings items to the customers from the kitchen.

so, here customer is a one application & kitchen is a another application, & waiter is worked as API between these two applications.

* What is REST: Its stands for REpresentational State Transfer.

⟶ It is a Architectural style, which allows system to take a wide range of data & it gives the data to the client at any time in the required format.

* popular API REST request format:

⟹ ① REST

② SOAP ⟶ (simple object Access protocol)

③ XML-RPC ⟶ (Rempote procedure call)

* popular API REST response formats:

⟹ ① REST

② XML-RPC

③ SOAP

④ JSON

⟹ nianan Rest Framework developed by Roy Fielding

* what is a RESTful web services:

⟹ A RESTful webservice is also called as RestFul web API.

→ It is a webservices to implemented by using HTTP method & it's principles of a REST.

here, we use a HTTP methods like (GET & POST)

→ It is a collection of resources with a four different Aspects;

① The base URI for the web service such as;

⟹ http://example.com/resources/ [uniform Resource—Identifier]

② The internet media type of the data supported by the web service, this is often json, xml, or YAml (yet-another Markup Language).

③ The set of operations supported by the web-services, using HTTP methods (Get, post, put, & Delete).

④ The API must be Hypertext driven.

* RESTFul web services [RestFul web API]:

⟹ To create a complete RESTFul web services, we should follows some constraints, which are as following;

① client-server: The client-server constraints works on the concept of that the client & server should be seperate from each other & allowed them to develope individually & independantly.

- In other words, we should be able to make changes to our mobile applications without impacting either the datastructure are the database design on the server.

→ At the same time, we should be able to modify the database or make changes to our server without impacting the our mobile applications.

→ It allows each applications grow & scale independently of the other & also allows our organizations to grow quickly & efficiently.

② Stateless: Roy fielding got inspiration from HTTP, so Http deflects In this constraints.

→ HTTP makes all client server interaction stateless.

→ stateless means no counting.

→ server will not store anything about latest http request which is made by client.

→ It will treat each & every request as new request.
→ server will not track any sessions & also history.

③ cache: A stateless API can increase the request overheads (buttons) by handeling large loads of incoming & outgoing calls.
→ A Rest API should be designe to store the cache data.

④ uniform interface: The uniform interface allows the client to talk to the server in a single Language independent of architecture backend of either.
→ This Interface should provide a unchanging, standardized, means of communicating between in client & server, such as using http with URI resources, & CRUD operations & JSON.

⑤ Layered system: As the name implies a Layered system is a system comprised of Layers with each Layer having a specific functionality & responsibility.
→ If we think of a model view Template framework, each Layer has it's own responsibilities, & models comprising how the data should be formed.
→ The Template focusing presentations & the view focusing on the incoming actions.
→ Each Layer has seperate but also interacts with the other.

⑥ code on demand (optional): This constraint is optional most of the time you will be sending the resources in form of XML or JSON.

* Installing Django Rest framework: To install Django rest framework, we should Install first python as well as Django.

⇒ python
Django
Django Rest Framework

⇒ pip install djangorestframework.

* Create a simple Rest API:
⇒ following are the steps for creating a simple Rest API;

• step1 : create a new with empty folder, to store the project.

—> open folder & type cmd to open command prompt with current locations.

=> E: restfolder>

—> create a project with name;

=> django-admin startproject studentpro;

—> change the path to the manage.py file & then create app with any name;

=> cd studentpro

=> python manage.py startapp studentapp.

—> goto mysql & create a database with name.

=> create database studentcpmdb;

=> use studentcpmdb;

—> open pycharm in restpro folder.

—> goto setting.py & configure db & Installed apps;

- we have to specify rest-framework under Installed apps regularly, beacuse it is a default app in restframework, & also we have to specify our appnames.

=> Installed APPS; [

:

'rest_framework',

'Studentapp',

]

—> goto databases option & configure the databases;

=> DATABASES = {

'default' : {

'Engine' : ------ mysql',

'Name' : 'studentcpmdb',

'user' : 'root',

'password' : 'root',

}}

—> goto models.py file & write below code;

==> class student (models. Model):
    sno = models. Integerfield ()
    sname = models. Charfield (max_length = 20)
    semail = models. Emailfield (max_length = 40)
    sage = models. Integerfield ()

    def __str__ (self):
        return self. sname.

--> goto admin.py file & write below code;

==> from . models import student

class Adminstudent (admin. ModelAdmin):
    list_display = ['sno', 'sname', 'semail', 'sage', 'sloc']
admin. site. register (student, Adminstudent)

--> goto views.py file & write below code;

==> from django. http JSONResponse
from . models import student
def studentlist (request):
    s = student. objects. all ()
    data = {"results": list (s. values ('sno',
        'sname',
        'semail',
        [redacted],
        [redacted] }})
        'sage'
    return JSONResponse (data)

--> goto urls.py file & write below code;

==> from studentapp import views
urlpattern = [
    url (-------),
    url (r'student', views. studentlist)
]

--> python manage.py runserver
--> copy the Ip address
--> open postman tool, to test API urls, & paste the url in postman tools.

→ we have to add 1 student to the url.

⇒ postman

http://127.0.0.1:8000/studnt


```
{"results": [
  {"sno": 1,
   "sname": "ramu",
   "semail": "ramu@gmail.com"
   "sfee": 1500,
   "sloc": "Hyd"},
  {
   "sno": 1,
   "sname": "RAJ",
   "semail": "raj@gmail.com",
   "sfee": 1200,
   "sloc": "pune"
  }
]}
```


→ This data was given through the admin site.

* Class based views : A view is a python function which takes request & gives response.

→ A view is a responsible to perform business logics.

→ views are two types generally;

① function based views; here we use def keyword.

② class based views; here we use a class keyword.

→ we can also write our APIViews by using class based views, rather than function based views.

→ This powerfull features allows us to reuse common functionality, & helps us to keep our code DRY [Don't Repeat yourself].

→ Class based views provide a alternative way to implements views instead of a functions.

→ They do not replace views, but they have certain difference & advantage, when compared to function based views.

① organization of code related a http method (Get, post, etc) can be implemented by seperate method instead of conditional branching.

② object oriented techniques, such as mixins (Inheritance) can be used in a reusable components.

→ A class based views allows us to respond to two different http request methods with a different class instance

O —> instead of with conditional branching inside single view functions.

* Syntax in Function based views;

—> def display (request):
    if request.method == 'post':
    elif request.method == 'Get':
    elif request.method == 'put':
    else request.method == 'Delete':

* Syntax in class based views;

—> class myclass():
    def get():
    def put():
    def 'post():
    def Delete():

* Django RestFramework (DRF):

—> It provide some APIView class, APIView class is a different from a regular view class in following ways;

① A request which is passed to the Handler method will be RestFramework request, not Django http request.

② The Handler methods, may return DRF response instead of Django Http response.

* Example by using APIView class:

=>
—> foldername: cbvfolder
—> projectname: cbvpro
—> appname : cbvapp
—> open cbvfolder in pycharm.
—> goto setting.py, to configure Installed Apps;

⟹ Installed Apps = [

    .
    .
    .

'rest_framework',
'cbvapp',
]

⟶ goto views.py file & write below code.

⟹ from rest_framework.views import APIViews
from rest_framework.response import Response
class Institute(APIViews):
    def get(self, request):
        databases = [
            {
            'oracle': 1000,
            'mysql': 2000
            }
        ]
        languages = [
            {
            'python': 1500,
            'django': 2000
            }
        ]
        return Response(
            {
            'databases': databases,
            'languages': languages
            }
)

⟶ goto urls.py file & write below code;

⟹ from django.conf.urls import url
from django.contrib import admin
from cbvapp import views
url patterns = [
    url [r'^admin/', admin.site.urls],
    url [r'^institute/', views.InstituteClass.as_views()],
]

⟶ run the server;

→ open postman tools & paste ip address & add /Institute.

* Get | http://127.0.0.1:8000/Institute | post → click on here.

→ it will display following format output;

⇒
```
"databases":[
      {
       "oracle":1000,
       "mysql":2000
      }
    ]
 "Languages":[
       {
        "python":1500,
        "Django":2000
       }
      ]
```

* Serializers: A serializers are used to convert model instance or querysets into native datatypes then, it will converted into json formats.

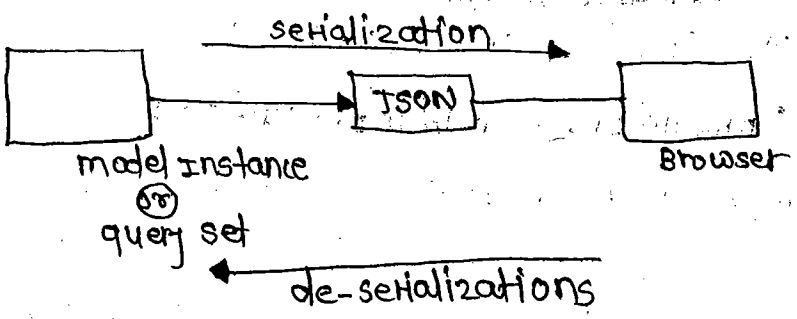
→ This process is called as "serializations".

→ The clients will use a client app's & then use convert JSON formats, into their required web pages.

→ serializers also supports "deserialization" which takes data from the browsers & then converts into JSON & then converts into model instance & query sets.

→ we create a new python name serialization.py to create a serializers & de-serializations.

⇒



⇒ ● By using Serializers.ModelSerializers ●

Step 1: foldername: product folder

→ project name: product data.

→ App name: productapp.

→ App Name: api

→ open product folder. in pycharm.

→ goto setting.py file & configure Installed_Apps;

⇒ Installed_Apps=[

'rest_framework',
'api',
'productapp',
]

→ goto productapp / models.py file, & write below code;

⇒ class product(models.Model):
    product_num = models.Integer
    product_name-

⇒ class productInfo(models.Model):
    pno = models.IntegerField()
    pname = models.charfield(max_Length=20)
    pcost = models.IntegerField()
    pdesc = models.Charfield(max_Length=40)
    def __str__(self):
        return self.pname

→ goto admin.py / productapp;

⇒ from. models import productInfo

class AdminproductInfo [redacted](admin.ModelAdmin):
    list_display=['pno','pname','pcost','pdesc']
admin.site.register(productInfo, AdminproductInfo)

→ goto api app/ & create a new file with name serializer.py

→ click on api → new → python file ⇒ serializer.py

→ open serializer file & write the below code;

⇒ from rest_framework serializers
from productapp import models

```
class productInfoSerializer(serializers.ModelSerializer):
    class Meta:
        fields=('__all__')
    model=models.productInfo
```

→ goto api/views.py file & write a below code;

```
=> from rest_framework import generics
   from . import serializers
   from productapp import models
   from . import serializers

   class Listproduct(generics.ListCreateAPIView)
       queryset=models.productInfo.object.all()
       generics_class=serializers.productInfoSerializer
```

→ goto api/urls.py file & write a below code;

```
=> from django.conf.urls import url, include
   from django.contrib import admin
   urlpatterns=[
       url(r'^admin/', admin.site.urls),
       url(r'^api/', include("api.urls")),
   ]
```

→ run makemigrations command.

```
=> python manage.py makemigrations
=> python manage.py migrate.
=> python manage.py createsuperuser.
=> python manage.py runserver.
```

→ copy Ip address login to admin site & enter product details.

=> change product info

pno : 10	pno : 11
pname: [redacted]	pname: Tab
pcost : 15000	pcost : 30000
	pdesc : [redacted]

→ open new tab & enter ip address along with /api.

=> 127.0.0.1:8000/api [post]

[get]

```
[
 {
"pno":10,
"pname":"Lap",
"pcost":15000,
"pdesc":"It is very Cheap"
 },
 {
"pno":11,
"pname":"Tab",
"pcost":30000,
"pdesc":"It is expence"
 },
]
```

[Raw data] [Html]

=> pno : [ ]
pname: [ ]
pcost : [ ]
pdesc : [ ] [post]

→ After giving data, if you clicking on post, they add existing data in a JSON format.
→ we can add either in raw data or html form
→ if we click on raw data form, it will show all fields as a keys with empty /null values.
→ we add the values & click on post, so that the data will post.

- By using serializers.Serializer :

=>

step1 : serfolder
→ project name: empprn

—> empapp

—> db name: rapi

—> open ser.folder in pycham

—> goto setting.pj file & configure Installed_Apps & also database;

—> goto models.pj file & write below code;

```
=> class Emp(models.Model):
       ename = models.charfield(max_length=40)
       email = models.Emailfield(max_length=80)
           def __str__(self):
               return self.ename
```

—> goto admin.pj file & write below code;

```
=> from .models import Emp

   class AdminEmp(admin.ModelAdmin):
       list_display ['ename', 'email']

   admin.site.register(Emp, AdminEmp)
```

—> create a python file in Empapp, as a serializers.pj file & write a below code;

```
=> from rest_framework import serializers
   class EmpSerializer(serializers.Serializer):
       ename = serializers.charfield(max_Length=40)
       email = serializers.EmailField(max_length=50)
```

—> goto views.pj file & write below code;

```
=> from django.shortcuts import render
   from rest_framework import status
   from rest_framework.response import Response
   from rest_framework.views import APIView
   from .models import Emp
   from.serializers import EmpSerializer

   Class Empview(APIView):
       def get(self, request):
           emp = Emp.objects.all()
```

```
        return Response(serializer.data)
    def post(self, request):
        serializer = EmpSerializer(data=request.data)
        if serializer.is_valid():
            ename = serializer.data.get('ename')
            email = serializer.data.get('email')
            msg = "Hello {}, your mailid is {}".format(ename,email)
            emp = Emp.object.create(ename=ename, email=email)
            return Response({'message':msg})
        else:
            return Response(serializer.errors, status=status.HTTP_
                                    400_BAD_REQUEST)
```

—> goto urls.py file & write a below code;

```
=> from empapp import views
   urlpatterns = [
        url(r'^admin/', admin.site.urls),
        url(r'^empl', views.Empview.as_view()),
   ]
```

—> Run the commands;

=> python manage.py makemigrations.
=> python manage.py migrate.
=> python manage.py createsuperuser.
=> python manage.py runserver.

—> Login to adminsite & enter two employees data Emp details;

=> Emp details.

ename: nani
email: nani@gmail.com

ename: Raj
email: Raj@gmail.com

→ open new table & enter ~~this~~ IP address along with /emp in the url path.

→ Then, it will display a code.

⇒

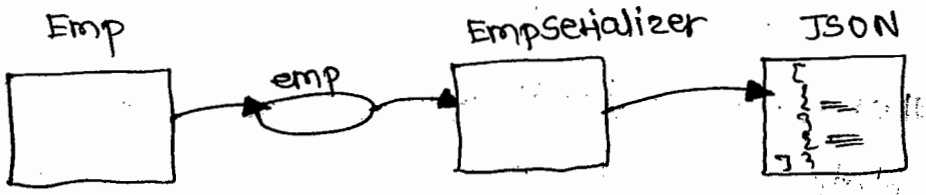
127.0.0.1:8000/emp

SEND | option | Get

Emp

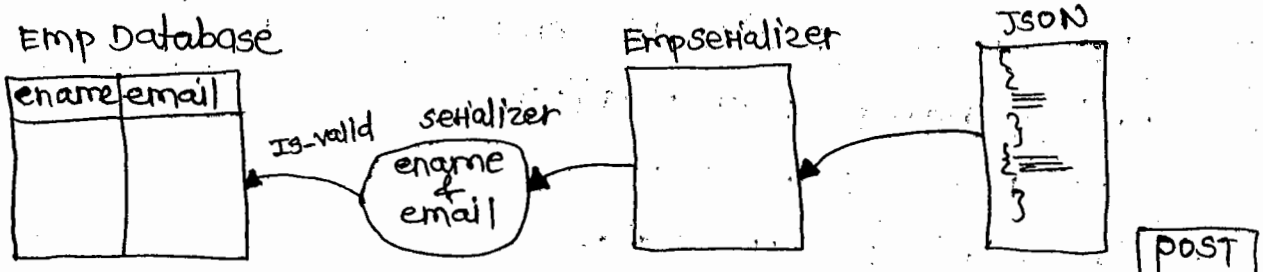
```
[
{"ename": "nani",
 "email": "nani@gmail.com"
},
{"ename": "Raj",
 "email": "Raj@gmail.com"
}
]
```

⇒ Serialization →



→ we can post a new employee data into a database;

⇒



← De-serialization

→ copy the nani data & paste in the content box & change & change the values & click on post button.

⇒

```
{
ename: 'nani'
email: ....
}
```

POST

→ If we click on post button, will save the data on the database & also message will be display on api

* performing CURD operations by using ModelSerializer:

=> If we want perform CURD operations in API then we should create two class based views.

--> In first class based view, we can get all resources & we can post one.

--> In second class based view, we can get one, Delete one or put one.

--> here, we can use a serializer.Serialize / serializer.ModelSerializer to perform CURD operations, but if we use serializer.ModelSerializer, then we can decrease the length of program in serializer, & also we can save time, when we have more number of fields in the model.

	(API)	(Django)
=> C	POST	create
U	PUT	Update
R	Get	Retrieve
D	Delete	Delete

* program:

=>

step 1: create folder as empapi folder.

--> project name : empapipro

--> app name : empapp

--> db name : apiempdb.

step 5 : goto setting.py & configure Installed app & db,

--> goto models.py file & write a below code;

=> from django.db import models

```
class Emp(models.Model):
    empid = models.IntegerField(primary_key=True)
```

⊗ salary = models.IntegerField(

salary = models.DecimalField(max_digits=10, decimal_places=2)

—> goto admin.py file & write a below code;

=> from .models import Emp

class AdminEmp(admin.ModelAdmin):
    list_display ['empid', 'empname', 'salary']

admin.site.register(Emp, AdminEmp)

—> create a file as, serializers.py file in app & write a below code;

=> from rest_framework import serializers
from .models import Emp

class EmpSerializer(serializers.ModelSerializer):
    class Meta:
        model = Emp
        fields = ('__all__')

—> goto views.py & write a below code;

=> from django.shortcuts import render
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from .models import Emp
from .serializers import EmpSerializer

class EmpView(APIView):
    def get(self, request):
        emps = Emp.objects.all()
        serializer = EmpSerializer(emps, many=True)
        return Response(serializer.data)

    def post(self, request):
        serializer = EmpSerializer(data=request.data)

```
        if serializer.is_valid():
            serializer.save()
            return Response({'message': 'new object is added to
                                Database'})
        else:
            return Response(serializer.errors, status=status.HTTP_400_
                                BAD_REQUEST)

class EmpDetails(APIView):
    def get(self, request, pk):
        e=Emp.objects.get(empid=pk)
        serializer=EmpSerializer(e)
        return Response(serializer.data)

    def post(self, request, pk):
        e=Emp.objects.get(empid=pk)
        serializer=EmpSerializer(e, data=request.data)

        if serializer.is_valid():
            serializer.save()
            return Response({'message': 'objects is successfully
                                updated'})
        else:
            return Response({'message': 'Record is not updated
                                successfully'})

    def delete(self, request, pk):
        e=Emp.objects.get(empid=pk)
        e.delete()
        return Response({'message': 'record deleted success
                                fully'})
```

→ create a urls.py file in a applications. & write code;

⇒
```
from django.conf.urls import url
from EmpApp import views
urlpatterns = [
```

```
    url(r'^emp/(?P<pk>[0-9]+)', views.EmpDetails.as_view()),
]
```

—> goto urls.py file in project level & write a below code.

=>

```
from django.conf.urls import include
urlpatterns=[
    url(-----),
    url(r'^api/', include(EmpApp.urls'))
]
```

—> Run the server & copy the ipaddress.

—> open postman tools, & paste the ipaddress in the postman url path;

—> If you specify http://127.0.0.1:8000/api/emp & select get option.

—> it will get all the data from the database.

—> all table ~~format~~ data will converted into JSON format & it displays.

=>

```
[
 {"empid":101,
  "empname":"saj",
  "salary":"20000.00"
 },
 {
  "empid":102,
  "empname":"Nani",
  "salary":"40,0000"
 },
]
```

—> specify http://127.0.0.1:8000/api/emp and select post potion & select Body part & add new one (post new one)

post

Send

=> {"empid":103,
"empname": "venkat",
"salary": "5000.00"
}.

=> ["message", "new object is added to database"]

—> NOTE: now we have 3 employees in the database.

—> specify http://127.0.0.1:8000/api/emp/101 & select get option.

—> now it displays 101 employee data in JSON format.

=> {
"empid": 101,
"empname": "sai",
"salary": "1000.00".
}

—> specify http://127.0.0.1:8000/api/emp/101 & select put option

=>

empid	ename	salary
101	sai	10000

—> In database table, empid 101 has some details like below.

—> NOW, I want update ename value & salary like below.

=>

empid	ename	salary
101	sairam	60000

—> so we take this new data in JSON format to update the existing data like below.

=> {
"empid": 101,
"empname": "sairam",

→ This new data will go as request to the put of inreplace the old data which already in the table.

→ specify http://127.0.0.1:8000/api/emp/101 & select Delete option,

→ Now it deletes employee data, whose empid is 101, the remaining data is like below,

```
=> [
    {
      "empid": 102,
      "empname": "Nani",
      "salary": "20000.00"
    },
    {
      "empid": 103,
      "empname": "venkat",
      "salary": 20000
    }
  ]
```

* Mixins: one of the big advantage of using the class based views is that, it allows us a to easily compose reuseable bits of behaviour.

→ The create/retrieve/update/delete operations the we have been using so far are going to be pretty similar for any model-backed API views we create.

- Those bits of common behaviour are implemented by using Rest framework mixins classes.

→ mixins classes provided the actions tha provided the basic view behaviour.

→ mixin classes provided actions methods rather then defining a handler methods.

→ The mixin classes can be imported by "rest_framework.mixins."

* Types of mixin classes:

1. ListModelMixin → getting all
2. RetrieveModelMixin → getting one
3. CreateModelMixin → post one
4. UpdateModelMixin → put one
5. DestroyModelMixin → Delete one.

1. ListModelMixin: It provides a .list(request, *args, **kwargs) method, that list out all available model instances.

2. RetrieveModelMixin: It provides a .retrieve(request, *args, **kwargs) method that returns an existing model instances as a response.

3. CreateModelMixin: It provides .create(request, *args, **kwargs) method, to post a new model instance & also it will save a new model instance

4. UpdateModelMixins: It provides .update(request, *args, **kwargs) method, it allows asa to modify the existing model instance, after modifying, it also save the model instance.

5. DestroyModelMixin: It provides .Destroy(request, *args, **kwargs) method, it allows to delete a the existing model instance.

=> foldername: mixinsfolder.

—> project name: mixins_pro.

—> app name : mixins_app.

—> open mixinsfolder in pycharm;

—> goto setting.py file & configure Installed apps;

=> [
'mixins_app',
'rest_framework',
]

—> goto models.py file & write a below code;

=>
```
class product(models.Model):
    product_id = models.IntegerField(primary_key=True)
    product_name = models.charfield(max_length=40)
    product_price = models.FloatField()
    product_color = models.charfield(max_length=20)
```

—> goto admin.py file & write below code;

=>
```
from .models import product
class Adminproduct(admin.ModelAdmin):
    list_display = ['product_id',
                    'product_name',
                    'product_price',
                    'product_color']
admin.site.register(product, Adminproduct)
```

—> create a serializers.py file & write below code;

=>
```
from rest_framework import serializers
from .models import product
class productserializer(serializers.ModelSerializer):
```

```
    class Meta:
        model = product
        fields = ('__all__')
```

=> goto views.py file & write a below code,

=>
```
from rest_framework import mixins
from rest_framework import serializers
from .serializers import productSerializer
from .models import product

class productList(mixins.ListModelMixin,
                  mixins.CreateModelMixin,
                  generics.GenericAPIView):
    queryset = product.object.all()
    serializers_class = productSerializer

    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs)
        return self.create(request, *args, **kwargs)


class productDetail(mixins.UpdateModelMixin,
                    mixins.DestroyModelMixin,
                    generics.GenericAPIView):

    queryset = product.objects.all()
    serializer_class = productSerializer

    def get(self, request, *args, **kwargs):
        return self.retrieve(request, *args, **kwargs)
    def put(self, request, *args, **kwargs):
        return self.update(request, *args, **kwargs)
```

```
    def destroy( self, request, *args, **kwargs):
        return self.destroy
```

→ create a urls.py file in application & write a below code;

```
from django.conf.urls import url
from . import views

urlpatterns = [
        url(r'^product/$', view.product_list.as_view()),
        url(r'^product/(?P<pk>[0-9]+)', views.product
            detail.as_view()),
    ]
```

→ goto project level urls.py file & write a below code;

```
from django.conf.urls import url, include

urlpatterns=[
    url('admin/', ---),
    url(r'^api/', includes( mixin_app.urls)),
]
```

=> python manage.py makemigrations.
=> python manage.py migrate.
=> python manage.py createsuperuser.
=> python manage.py runserver.

→ login to the adminsite & enter some product details.

→ open new tab & enter ipaddress along with /api/product

→ 127.0.0.1:8000/api/product/

• NOTE: here, we can check all operations, like previous example.

* Different Handler methods are:

1. get()
2. post()
3. put()
4. delete()

* Different Action methods are:

1. list() ——> ListModelMixins.
2. retrieve() ——> RetrieveModelMixin.
3. create() ——> CreateModelMixin.
4. update() ——> UpdateModelMixin,
5. Destroy() ——> DestroyModelMixin,

—> Here, we use both Handler & Action methods in class based views to handle corresponding request like get(), post(), put(), & Delete() method.

—> The same requests are handeled by using conditional statements in function based views.

* Function based views: We write the program to handled all types of request based on if statement.

—> In class based views, we write a specific handler method for specific request, but in function based views, based on we write if statements.

=> foldername: Function based.
—> project name: functionbase
—> app name: function_based_App.

—> goto setting.py file & specify Installed app & rest

→ goto models.py file & write a below code;

```
=> class product(models.Model):
       product_id = models.IntegerField(primary_key=True)
       product_name=models.CharField(max_length=20)
       product_price = models.DecimalField(max_digits=20,
                                           decimal_places=2)
       product_color = models.CharField(max_length=40)
```

→ create a serializers.py file in app & write a below;

```
=> from rest_framework import serializers
   from .models import product

   class productSerialzer(serializers.ModelSerializer):
       class Meta:
           Model = product
           fields = ['product_id',
                     'product_name',
                     'product_price',
                     'product_color']
```

→ goto views.py file & write a below code;

```
=> from rest_framework import status
   from rest_framework.decorators import api_view
   from rest_framework.response import Response
   from .serializers import productSerializer
   from .models import product

   @api_view(['GET', 'POST'])
   def productList(request):
       if request.method == 'GET':
           products = product.objects.all()
       serializer = productSerializer(products, many=True)
       return Response(serializer.data)
       elif request.method == 'POST':
           serializer = productSerializer(data=request.data)
```

```
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.HTTP_201_CREATED)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

```
@api_view(['GET', 'PUT', 'DELETE'])
def productDetail(request, pk):
    try:
        products = product.objects.get(pk=pk)
    except product.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)
    if request.method == 'GET':
        serializer = productSerializer(products)
        return Response(serializer.data)
    elif request.method == 'PUT':
        serializer = productserializer(products, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
    elif request.method == 'DELETE':
        products.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

→ create urls.py file in application & write below;

```
=> from django.conf.urls import url
   from . import views
   urlpatterns = [
       url(r'^product/$', views.productList),
       url(r'^product/(?P<pk>[0-9]+)', views.productdetail)
   ]
```

—> goto project level urls.py file & write a below code;

=> from ------ admin

from django.conf.urls import url, include

urlpatterns=[

url(------),

url(r'^api/', include('function_Based_App.urls')),

]

—> goto admin file

=> from models import product

class Adminproduct(admin.ModelAdmin):

list_display ['product_id',
'product_name',
'product_price',
'product_color']

admin.site.register (product, Adminproduct)

—> rem makemigrations

—> python manage.py migrate.

—> python manage.py createsuperuser.

—> python manage.py runserver.

* View sets: Generally we write logic to create a end point urls, by using class based views in rest framework.

So, django rest framework supports to create end point urls;

① APIViews

② View-sets

—> In APIview, we create a two class based views, one is for non-primary key based class, & the other one is primary key based class, but in viewsets we create only one class based views for both primary & non-primary key based.

—> In APIViews we use Handler methods, like get(), post(), put(), Delete(), but in Viewsets we used Action methods like list(), retrieve(), create(), update(), destroy().

—> In APIviews we map urls manually, but in viewsets we don't map urls manually, we use router to map urls.

—> create a project.

—> project name: viewsetpro

—> App name: viewsetapp

—> db name: viewsetdb

—> goto settings.py & configure Installed_Apps & Database.

—> goto models.py & write the below code;

```
=> Class Feedback(models.Model):
       wer_id = models.IntegerField()
       name = models.CharField(max_length=200,
                               help_text='Name of the
                                          sender')
       email = models.EmailField(max_length=200, unique
                                          =True)
       subject = models.CharField(max_length=200)
       message = models.TextField()
       created = models.DateTimeField(auto_now_add=
       modified = models.DateTimeField(auto_now=True)
                                          True)
```

```
    def __str__(self):
        return self.name + "-" + self.email
```

—> goto admin.py file & write a below code;

```
=> from django.contrib import admin
   from .models import Feedback
   class AdminFeedback(admin.ModelAdmin):
       list_display = ['user_id',
                       'name',
                       'email',
                       'subject',
                       'message',
                       'created', 'modified']
   admin.site.register(Feedback, AdminFeedback)
```

—> create a serializers.py file in applications;

```
=> from rest_framework import serializers
   from .models import Feedback
   class FeedbackSerializer(serializers.ModelSerializer):
       class Meta:
           model = Feedback
           fields = ('__all__')
```

—> goto views.py file & write below code;

```
=> from rest_framework.response import Response
   from rest_framework import status
   from rest_framework import viewsets
   from .models import Feedback
   from .serializers import FeedbackSerializer
```

```
class viewset_feedback(viewsets.Viewset):
    def list
```

```
Class viewset_feedback(viewsets.Viewset):
    def list (self, request):
        queryset = Feedback.objects.all()
        serializer = FeedbackSerializer (queryset, many=True)
        return Response(serializer.data)

    def create(self, request):
        serializer = FeedbackSerializer (data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response (serializer.data, status=status.
                                                HTTP_201_CREATED
        else:
```

```
    Return Response(serializer.errors, status= status.HTTP_
                                        400_BAD_REQUEST)
def retrieve (self, request, pk):
    queryset = Feedback.objects.get(pk=pk)
    serializer= FeedbackSerializer(queryset, data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serialize.data, status= status.HTTP_
                                        201_CREATED)
    else:
        return Response(serializer.errors, status=status.HTTP_
                                        400_BAD_REQUEST)
def destroy(self, request, pk):
    queryset= feedback.objects.get(pk=pk)
    queryset.delete()
    festum Response(status=status.HTTP_204_NO_CONTENT)
```

→ create a urls.py file in application & write a below;

```
=> from django.conf.urls import url
   from django.conf.urls import include
   from . import views
   from rest_framework.routers import DefaultRouter

   router= DefaultRouter()
   router.register('viewset', views.ViewSet_feedback, base_
                                name='viewset')

   urlpatterns=[
       url(r'^', include(router.urls))
   ]
```

→ goto urls.py file in a project level & write code;

```
=> from django.contrib import admin
   from django.conf.urls import url, include
```

```
urlpatterns=[
    url(------)
    url(r'^feedback/', include(Viewset_App.urls'))
```

—> Run the commands;

=> makemigrations.
=> migrate.
=> createsuperuser.
=> Runserver.

* **ModelViewset**: ModelViewset class is a derived from rest_framework.

—> It contains list(), Retrieve(), update(), create(), & destroy() Internally.

—> It has two attributes: ① queryset ② serializer_class.

—> we load all model data into queryset attribute & serializer_class into serializer_class attribute.

—> In viewset class, we define all '5' methods manually but in Modelviewset class, those are default method.

—> step 1: foldername mvsfolder.
—> project name: mvspro
—> app name: mvsapp.
—> db name: mvs_db

—> goto setting.py file & configure installed apps & database;

—> goto models.py file & write below code;

```
=> class Emp(models.Model):
    empid = models.IntegerField(primary_key=True)
    empname = models.CharField(max_length=20)
    email = models.EmailField(max_length=40)
    salary = models.DecimalField(max_digits=10, decimal_
                                 places=2)
```

→ goto admin.py file & write below code;

=> from .models import Emp

```
class AdminEmp(admin.ModelAdmin):
    list_display=['empid',
                  'empname',
                  'email',
                  'salary',]

admin.site.register(Emp, AdminEmp)
```

→ create serializer.py file & write a below code;

=> from rest_framework import serializers
from .models import Emp

```
class EmpSerializer(serializers.ModelSerializer):
    class Meta:
        model=Emp
        fields=('__all__')
```

→ goto view.py file & write below code;

=> from rest_framework import viewsets
from ModelViewset_App.models import Emp
from .serializers import EmpSerializer

```
class EmpViewset(viewsets.ModelViewset):
    queryset=Emp.object.all()
    serializer_class=EmpSerializer.
```

→ goto urls.py file in applications & write code;

=> from django.conf.urls import url, include
from ModelViewset_App import views
from rest_framework.routers import DefaultRouter
router=DefaultRouter()

```
router.register('emp-viewset', views.EmpViewSet())
urlpatterns=[
    url(r'', include(router.urls))
]
```

→ goto urls.py file in project levels & write below:

Admin

=> from django.conf.urls import url, include.

```
urlpatterns=[
    url('admin/', admin.site.urls),
    url(r'^api', include('modelviewset_App.urls'))
]
```

→ Run the commands

=> makemigrations.
=> migrate.
=> createsuperuser.
=> runserver. with api

* Security: Generally we provide a security for our data, to avoide a other people haking the data.

→ we have seen multiple example, which are not having any kind of security, that means everyone who knows end point url, they can hack our data.

→ If we have user & password for any end point url, then others can not perform CURD operations.

→ we provide in a two ways;

1. Authentication.
2. Authorization (permissions).

→ Authentication is a used to check wheather he entered valid username & password.

→ Authentications are diffamt types;

① Basic Authentication ② session ③ Token.

→ Authorization is to check what permissions this specified user has, Authorization or permission are differept types;

① IsAuthenticated

② IsAdmin

③ IsAdminOrReadonly.

① Basic Authentication: It is also called as Default Authentication.

→ In Basic Authentication, after entering along with Ip address immediately it will open default window which ask username & password.

→ If we specify username & password then only it will open end point url, if the username or password are invalid then it will redirect to same window.

=> project name: BAproo.

→ Appname: BAapp.

→ dbname: badb.

→ goto settings.py file & configure database & Installed apps.

→ goto models.py file & create model Emp with empid, ename, esalary fields.

→ goto admin.py file & write the code.

→ create a serializers.py file & create a serializer with name EmpSerializer.

—> goto views.py file & write a below code;

```
from .serializers import EmpSerializer
from .models import Emp
from rest_framework import viewsets
from rest_framework.authentication import
                                BasicAuthentication
from rest_framework.permissions import IsAuthenti-
                                cated.

class EmpViewset2(viewsets.ModelViewset):

    authentication_classes=(BasicAuthentication,)
    permission_classes=(IsAuthenticated,)

    queryset = Emp.objects.all()
    serializer_class = EmpSerializer.
```

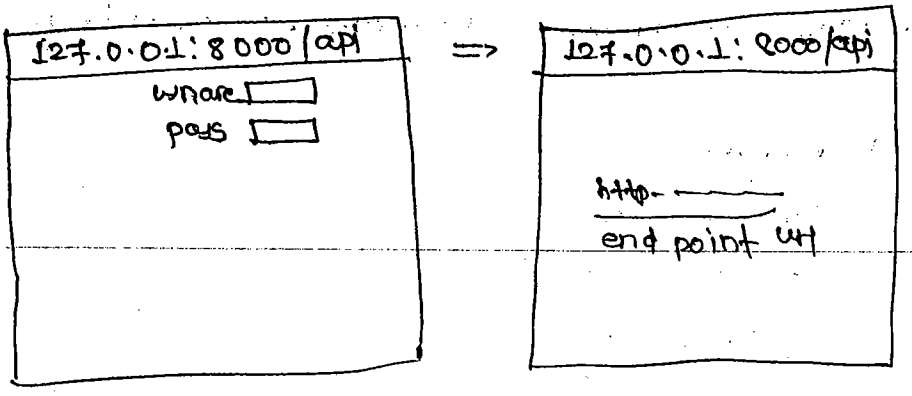
—> create a urls.py in applications;

```
=> from django.conf.urls import url, include
   from rest_framework.routers import DefaultRouter
   from .views import EmpViewSet2

   router= DefaultRouter()
   router.register('emp_viewset', EmpViewSet2, base_name
                                = 'emp_viewset2')

   urlpatterns=[
        url(r'', include(router.urls))

   ]
```

—> run the server;

—> copy the IP address & paste in the browser along with a /api, if we click enter then a displays small sign in form for username & password,

- after giving username & password, if we click on sign in button it will show as a endpoint url, like below;



*(2) session Authentication:

—> project name: sessionpro2
App name: sessionapp2
dbname: sadb2.

—> goto settings.py file & configure database & Installed apps.

—> goto models.py file & create Emp model with (id, name, & salary fields).

—> goto admin.py file & configure.
—> serializers.py file.
—> goto views.py file & write a below code;
~~=> create serializers.py file.~~
=> from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login
from .serializers import EmpSerializers
from .models import Emp
from rest_framework import viewsets
from rest_framework.authentication import SessionAuthentication.
from rest_framework.permissions import IsAuthenticated.

```
def login_form(request):
    return render(request, 'login.html')
def login_user(request):
    username = request.POST['uname']
    password = request.POST['pwd']
```

```
user= authenticate (username= username, password= password)
print(user)
If user is not None:
        login(request, user)
        return redirect('/api')
else: return redirect('/auth')

class EmpViewset2 (viewsets. ModelViewSet):
    authentication_classes =( SessionAuthentication,)
    permission_classes =( IsAuthenticated,)
    queryset= Emp. object.all()
    serializer_class = EmpSerializer
```

→ serializers.py

```
⇒ from rest_framework import serializers
  from .models import Emp

  class EmpSerializer(serializers. ModelSerializer):
      Class Meta:
          model= Emp
          fields= ('empid', 'empname', 'salary')
```

→ create a html file with name login.html in a templates folder;

```
⇒ <body>
    <form action="/loginuser/" method="post">
      {% csrf_token %}
      username : < input type="text" name="uname">
      password : < input type='password' name="pwd"><br><br>
          <input type="submit" value="submit">
    </form>
  </body>
```

→ create a urls.py file in applications;

```
=> from django.conf.urls import url, include
   from rest_framework.routers import DefaultRouter
   from .views import EmpViewSet2

   router= DefaultRouter()
   router.register('emp_viewset', EmpViewSet2, base_name=
                                                'emp_viewset2')
   urlpatterns=[
       url(r'', include(router.urls))
     ]
```

→ goto urls.py in project;

```
=> from django.contrib import admin
   from django.conf.urls import url, include
   from Basic_Authentication_App import views
   url patterns=[
      url('auth/', views.login_form),
      url('loginuser/', views.login_user),
      url(r'napi/', include('Basic_Authentication_App.urls')),
      url('admin/'),
    ]
```

→ makemigrations.
→ migrate.
→ createsuperuser. (password.)
→ runserver, copy the url & paste in the browser.

*

* (5es) Token based Authentication: It is a different from basic & session authentication.

- A token based Authentication we generate a token (random string value), for every user.

-> if any user wants to perform CURD operation on any end point url then, we should use username & password with along token.

- After creating a superuser then we login to the admin site with superuser & password.

-> we can create a one or more users, by using users default table.

-> After creating user then we generate a token to a user.

-> If any user wants any operation then he will use a random string.

-> we use a default app name 'rest_framework.authtoken' & 'tokenauth'. & under Installed apps.

==> project name: TokenAuthentication_project.

--> App name: - TokenAuthentication_App.

--> db name: tokenauthentication_db

--> goto setting.py file & configure a database & Installed apps.

```
==> Installed_APPS=[
    '
    '
    '
    'TokenAuthentication_App',
    'tokenauth',
    'rest_framework',
    'rest_framework_authtoken',
    ]
```

→ goto models.py file & write below code:

```
=> class Emp(models.Model):
       empid = models.IntegerField(primary_key=True)
       empname = models.CharField(max_length=200)
       empsal = models.DecimalField(max_digits=10, decimal_
                                    places=2)
       created = models.DateTimeField(auto_now_add=True)
       modified = models.DateTimeField(auto_now=True).
```

→ goto admin.py file & write below code:

```
=> from TokenAuthentication_App.models import Emp

   class EmpAdmin(admin.ModelAdmin):
       list_display = ['empid',
                       'empname',
                       'empsal',
                       'created',
                       'modified']
admin.site.register(Emp, EmpAdmin)
```

→ create a serializer.py files & write below code:

```
=> from rest_framework import serializers
   from .models import Emp

   class EmpSerializer(serializers.ModelSerializer):
       class Meta:
           model = Emp
           fields = ('__all__')
```

→ goto views.py file & write below code:

```
=> From django.shortcuts import render, redirect
   from .serializers import EmpSerializer
   From .models import Emp
```

```
from rest_framework import viewsets
from django.contrib.auth import authenticate, login

def login_form(request):
    return render(request, 'login.html')

def login_user(request):
    username= request.post('uname')
    password = request.post ['pwd']
    user= authenticate(username= username, password=
                                        password)
    print(user)
    print("===, ===")
    if user is not None:
        login (request, user)
        return redirect ('/api')
    else:
        return redirect ('/auth')

class EmpViewset (viewsets. ModelViewset):
    queryset= Emp. object. all()
    serializer_class= EmpSerializer
```

→ create html file with name login.html;

```
==> <form action="/loginuser/" method="post">
        {% csrf_token %}
        User Name: <input type="text" name="uname">
                                    <br> <br>
        password: <input type="password" name="pwd">
                                    <br> <br>
        <input type="submit" value="submit">
</form>
```

→ create urls.py in application level ;

=> from django.conf.urls import url, include
from TokenAuthentication_App import views
from rest_framework.routers import DefaultRouter

router = DefaultRouter()
router.register('emp-viewset', views.EmpViewset)

urlpatterns= [
    url(r'', include(router.urls))
]

→ create urls.py in project ;

=> from django.contrib import admin
from django.conf.urls import url, include
from TokenAuthentication_App import views as myviews
from rest_framework.authtoken import views

urlpatterns =[
    url('----'),
    url(r'auth/', myviews.login_form),
    url(r'loginuser/', myviews.login_user),
    url(r'api/', include('TokenAuthentication_App.urls')),
    url(r'gettoken/$', views.obtain_auth_token),
]

→ Run makemigrations.
→ migrate.
→ createsuperuser.
→ runserver

* Status codes :

① HTTP_200_OK : It is standard Response for successful HTTP resquests.

② HTTP_201_CREATED : The request has been fulfilled, resulting in the creation of a new resource.

③ HTTP_202_ACCEPTED : The request has been accepted, but, the processing has not been completed.

④ HTTP_204_NO_content : The server successfully processed the request, but it's not returning any content.

⑤ HTTP_205_Resent_content : The server successfully processed the request but it is not returning any content unlike 204 response, this response requires that the requestor reset the document view.

⑥ HTTP_400_Bad_Request : The server can not process the request due to any error.

⑦ HTTP_401_Unauthorized : 401 semantically means unauthorized "or unauthenicated", i.e. the user does not have the necessary credentials.

⑧ HTTP_403_forbidden : The request was valid, but the server is refusing actions.
- the user might not have the necessary permissions for a resource, or may need an account of some sort.

⑨ HTTP_404_NOT_Found : The request resource could not be found, but may be available in the future.

⑩ HTTP_500_internal_server_error : It is generic error message, given when an unexcepted conditions was encountered & no more specific message