

Viewssets :-

- * DRF providing "viewsets" module. This module providing several classes to performing the database operations
- * Basic CRUD operations if we want to implement then we should go for viewsets development
- * By using viewsets we can develop applications very fastly and easily then the APIViews
- * Here we are going to develop only one class for all crud operations. It can do both id & non-id based operations
- * We are not configuring the urls (~~automatically~~) manually. Router module is going to take care of creating the url patterns instead of programmers
- * Regular urls are not allowed in the Viewset concept
- * Regular urls are applicable only in APIView concept
- * By using the Viewset concept most of the business logics are done by internally automatically
- * DRF allows us to combine the logics for a set of related views in a single class called Viewsets. This module providing two classes
 1. Viewsets, ViewSet
 2. Viewsets, ModelViewSet

Viewsets • ViewSet :-

- * By using ViewSet class we write all different methods for handling different incoming request
 - * we need to write the separate logics in each method respectively
 - * ViewSet class allows writing all '5' method logics in a single view
- Eg:- `class viewName(ViewSet):` `def list(self, request):` `<logics>`

Eg:- class viewname (ViewSet):

def list (self, request):

<logics>

def create (self, request):

<logics>

* Here all method names are action methods like list, create, update, ...

Viewsets, ModelViewSet:

* By using ModelViewSet class we don't need to write different handler methods separately with required logics, bcz ModelViewSet can handle all request methods itself automatically with required logics

* Here we required only one class with queryset, Serializer class

Eg:- class Viewname (ViewSet, ModelViewSet):

queryset = modelname.objects.all()

serializer_class = user defined serializer class

Routers:-

* To creating the "model viewset" class based view related url, we are going to use routers concept but not general url's

* using general url's to creating the id based url we need to follow the regular expressions

* By using routers concept we no need to follow regular expressions we are going to create one normal string based url which can do both id & non-id based operations

* To creating the string url we are following routers procedure

Step-1:- importing the DefaultRouter class from routers module

from rest-framework routers import DefaultRouter

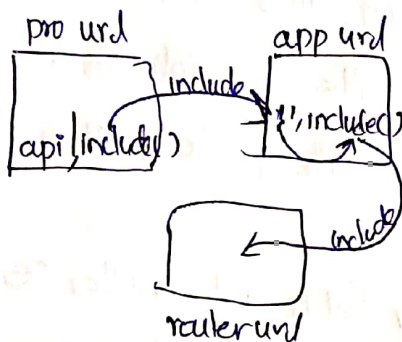
step-2:- create the object for DefaultRouter class

router = DefaultRouter()

step-3:- create any string based name into router object by using register method as a first parameter. Register() method can take our view name as a second parameter

router.register('emp', views.viewname)

* To executing this router url we need to create one url inside urlpatterns section. By using include() we will go to from this urlpatterns to router url section



Project:-

write a project to performing the database operations by using ModelViewset concept

step:- copy paste all files from previous programme except views.py & applevel urls.py

views.py:-

from models import emp

from serializer import empserializer

from rest-framework import Viewsets

class Empmodelviewset (Viewsets.ModelViewset):

queryset = emp.objects.all()

serializer_class = empserializer


```

open applerel urls.py
from django.urls import path, include
from mixinsapp import views
from rest_framework.routers import DefaultRouter
router = DefaultRouter()
router.register('emp', views.empmodelviewset)
urlpatterns = [
    path('', include(router.urls))
]

```

Nested Serializer

The process of converting database instance into dict type and then converted into JSON data type is called serialization

- ⇒ A serializer is writing inside another serializer class then it is called Nested serializer
- ⇒ Some times we are using one serializer fields into another serializer also then to access all fields at a time by sending one request
- ⇒ If we are using Normal serializer concepts then only we will access one serializer class fields only
- ⇒ If we want to access 2 serializer class fields then we should use Nested serializer concept
- ⇒ Generally when we are making the relationship between the table then this nested serializer concept we can use

Why should go for Nested serializer concept:

when we are sending one request for accessing the parent information then along with the parent information if you want to

access child information also then we should go for Nested serializer concept

Ex:- we have 2 model classes like author and book
for author we have 2 fields
→ first name
→ last name
→ subject
for book model we have fields like
→ title
→ release date
→ rating
→ author

⇒ Book model contains author field this is acting like a foreign key of Author model

⇒ To make the relation between these 2 models we will use foreign key field inside child model (Book)

⇒ To get the serialization effect between the both tables we should use "related_name" attribute with some user defined value as a parameter of foreign key value

Ex:- class Author (models.Model)

first name =

last name =

subject =

class Book (models.Model)

title =

rating =

release date

author = models.ForeignKey (Author, on_delete = models.
CASCADE related_name = "book - By - author")

for both models required serializer classes like Author serializer
Book serializer

⇒ whenever we are sending the request to Author information

Then compulsory this author returns books information also
I want to display

Then for this child serializer object inside parent serializer
class representing assigning books by Author value of related-name

```
class AuthorSerializer()
```

```
class Meta:
```

```
model =
```

```
fields = 'o--all--'
```

```
books_by_author = BookSerializer(many=True, read-only=True)
```

```
class BookSerializer()
```

```
class Meta:
```

```
model =
```

```
fields =
```

⇒ Because of books_by_author field inside parent serializer we
will get all child serializer class fields inside parent
serializer

Program:-

project name: Nested-serializersproject

app name: Nested-serializers-App

db : Nested-serializersdb

open settings.py file add our application name, rest-framework

inside INSTALLED_APPS section

Configure the database details under the database section
in settings.py file

open models.py file

```
from django.db import models
```

```
class Author(models.Model):
```

```
    first_name = models.CharField(max_length=100)
```

```
    last_name = models.CharField(max_length=100)
```

```
    subject = models.CharField(max_length=100)
```

```
    def __str__(self):
```

```
        return self.first_name
```

```
class Book(models.Model):
```

```
    title = models.CharField(max_length=100)
```

```
    author = models.ForeignKey(Author,
                                on_delete=models.CASCADE,
                                related_name='book-by-author')
```

```
    released_date = models.DateField()
```

```
    rating = models.IntegerField()
```

```
    def __str__(self):
```

```
        return self.title
```

open serializer.py

```
from rest_framework.serializers import ModelSerializer
```

```
from NestedSerializerApp.models import Author, Book
```

```
class BookSerializer(ModelSerializer):
```

```
    class Meta:
```

```
        model = Book
```

```
        fields = '__all__'
```



```
class AuthorSerializer (ModelSerializer):  
    books-by-author = BookSerializer (read-only=True, many=True)
```

```
class Meta:
```

```
    model = Author
```

```
    fields = '__all__'
```

Open views.py file and write below code:-

```
from Nested_Serializers_App.models import Author, Book  
from Nested_Serializers_App.serializers import AuthorSerializer, Book  
Serializer
```

```
from rest_framework import generics
```

```
class AuthorListView (generics.ListCreateAPIView):
```

```
    queryset = author.objects.all()
```

```
    serializer_class = AuthorSerializer
```

```
class AuthorDetailView (generics.RetrieveUpdateDestroyAPIView)
```

```
    queryset = author.objects.all()
```

```
    serializer_class = AuthorSerializer
```

```
class BookListView (generics.ListCreateAPIView):
```

```
    queryset = book.objects.all()
```

```
    serializer_class = BookSerializer
```

```
class BookDetailView (queryset = serializer_class = Book
```

```
    queryset = Books.objects.all()
```

```
    serializer_class = BookSerializer
```


open project level urls.py and write below code

from django.contrib import admin

from django.urls import path

from django.urls import url

from Nested-serializers-App import Views

urlpatterns = [

path('admin/', admin.site.urls),

url(r'^author-api/\$', views.AuthorListView.as_view()),

url(r'^author-api/(?p<pk>\d+)/\$', views.AuthorDetailView.as_view()),

url(r'^book-api/\$', views.BookListView.as_view()),

url(r'^book-api/(?p<pk>\d+)/\$', views.BookDetailView.as_view()),

]

⇒ Execute makemigrations, migrate, runserver command

127.0.0.1:9600/author-api/

{ "id": 6

"book-by-author": [

{ "id": 5,

"title": "Java",

"release_date": "2019-12-17",

"rating": 3,

"author": 6

}]

"first_name": "Ravi",

"last_name": "Kumar 123",

"subject": "HTML"