Django REST Framework Serializers Concept

1. What is serialization ?

- Serializers are used to convert the querysets and model instances into native Python datatypes(Dict) and then it can be easily rendered into JSON data type. this process is called serialization.
  QuerySet Data ------>> Dict Data ------>> JSON Data

2. What is deserialization ?

- Serializers also provide deserialization concept, which allows JSON data into complex or native datatypes(dict) and then it can be converted into querysets and models instances.

  JSON Data ----->> Dict Data ----->> QuerySet Data

- API stands per Application Programming Interface which accepts a user input to performing the required operations.
- For example submitting a new data , which is translated from HTML into Json format then converted into our Django model.

Q) How many ways we will perform serialization ?

There are so many ways available to perform the serialization,

For example,

- By using python in-built module "json" .
  - json.dumps(data)
  - json.loads(data)
- By using django's serialize() function.
  - from django.core.serializers import serialize --->> method.
- By using rest_framework serializers module
  - from rest_framework import serializers
- Django REST Framework is providing one module which name as "serializers" , By using this module we can convert one type of data into another type more easily.

Serializers module proving several classes to do this process like

1. serializers.Serialzer

2. serializers.ModelSerialzer

- The "serializers" module in Django REST framework work very similarly to Django's "forms" module like "Form" and "ModelForm" classes.
- serializers module provide a "Serializer" class which gives you a powerful, generic way to control the output of your responses, as well as a "ModelSerializer" class which provides a useful shortcut for creating serializers that deal with model instances and query sets.

Django follows MVT architecture where

- M stands per Models which is used for interacting with application database.
- V stands per Views which is used for doing application business logics.
- T stands per Templates which is used for doing Presentation logics of applications.

- A traditional Django app needs a dedicated url, view and template to translate information from the database onto a webpage.

Django REST Framework follows MVS architecture where

- **M** stands per **Models** which is used for interacting with application database.
- **V** stands per **Views** which is used for doing application business logics.
- **S** stands per **Serialzers** which is used for Transfering data from our application into another applications.
- In Django REST Framework we need a url, view and a serializer. But not Template.
- The **url** controls access to the API endpoints,
- **views** control the logic of the data being sent, and
- **serializer** performs the magic of converting our information into a format suitable for transmission over the internet, like JSON type.
- A normal webpage requires HTML, CSS, and JavaScript (usually). But our API is only sending data in the JSON format, so No HTML and No CSS, Just we need only data.
- Django REST Framework is providing one module which name as "serializers" to convert one type data into another type of data.

This "serializers" module providing several classes like

1. serilizers.Serializer
2. serilizers.ModelSerializer

- Note: Create one python file which name as "serializers.py" inside our application name.
- Open this userdefined "serializers.py" file and create required serializers code.

Procedure for serilizers.ModelSerializer

1. Importing django rest framework related "serializers" module.
2. Importing required models names here
3. Creating required userdefined serializer class by using predefined serializer class.
4. Create a Meta class as nested class to the current class and represent what is model name using "model" attribute and model fields using "fields" attribute.

Code:

```
from rest_framework  import  serialzers
from appname.models  import  model_name
class UserdefinedSerializer(serialzers.ModelSeializer):
    class  Meta:
        model = model_name
        fields = '__all__'
```

Procedure for serializers.Serializer

- Importing django rest framework related "serializers" module.
- Creating required userdefined serializer class by using predefined serializer class.
- Creating required Model class "fields" along with required filed data type classes.

code:

```
from rest_framework  import  serialzers

class UserdefinedSerializer(serialzers.ModelSeializer):

    eno    = serializers.IntergerField()

    ename  = serializers.CharField()

    salary = serializers.FloatField()
```

- Note: If we are using normal "Serializer" class then we need to represent all Model class related fields inside userdefined serializer class along with serializers filed data types.
- If we are using serializers.Serializers class then inside views we need to access all the serializers fields data and we need to store in separate variables.
- Then we need to store all the variables data in a table by using Model class name.
- Note : If we are using serializers.ModelSerializers class then no deed of representing all the model class fields as serializers fileds. Just we are representing what is model name and what are model filed names.
- If we are using serializers.ModelSerializers class then inside views we No need to access all the serializers fields data manuvally and we No need to store in separate variables.
- directly we can access all fields data and store into database tables automatically using save().

Conclusion :

- So compared to serializers.Serializer class , serializers.ModelSerializer is having more benifits. It is reducing lot of burden on developers while developing coding.

How to Creating Models code in Models.py

models.py procedure:

- importing the models module
- create the userdefined model class by using predefind model class
- craete the required userdefined model class fields along with required field datatypes

code:

```
from  django.db   import  models

class Employee(models.Model):

   eno    = models.IntegerField()

   ename  = models.CharField(....)

   salary = models.FloatField()
```