# CRUD Operations using Django Rest Framework(DRF)

## Introduction

In the world of web development, creating, reading, updating, and deleting data are fundamental operations that underpin the functionality of almost every application. Whether you're building a simple blog, a complex e-commerce platform, or a cutting-edge social media app, the ability to perform these operations efficiently is crucial. That's where Django Rest Framework (DRF) comes into play.

## Why Django Rest Framework?


CRUD, which stands for Create, Read, Update, and Delete, represents the core operations required to manage data in any web application. DRF, an extension to the popular Django web framework, empowers developers to effortlessly implement these operations while adhering to RESTful principles. It simplifies the process of building robust and scalable web APIs, making it the go-to choice for many developers.

### Serializers in DRF

The serializers in the REST framework work very similarly to Django's Form and ModelForm classes. We 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 querysets.

- **Functionalities of Serializers**

    1. **Data Validation:** Serializers validate incoming data, ensuring it adheres to predefined rules and constraints, helping maintain data integrity.

    2. **Data Conversion:** They convert complex data types, like database querysets or model instances, into JSON or other formats for API responses.

    3. **Deserialization:** Serializers reverse the process, taking incoming data (e.g., JSON) and converting it into complex Python data types, making it suitable for processing and saving to a database.

    4. **User Input Handling:** Serializers are often used for handling user input in forms and API requests, making it easy to bind incoming data to application logic.

    5. **DRF Views Integration:** In DRF, serializers are tightly integrated with views, simplifying the process of creating, updating, and retrieving data from APIs.

    6. **Nested Data Handling:** Serializers can handle complex, nested data structures, making them suitable for APIs that deal with related objects or hierarchical data.

    7. **Customization:** Developers can customize serializers to control how data is serialized, deserialized, and validated, allowing for fine-grained control over API behavior.

## Model Serializers

A model serializer in Django Rest Framework (DRF) is a powerful tool for simplifying the process of serializing and deserializing complex data models. It automatically generates serializers based on model definitions, reducing the need for manual coding.

## Views in DRF

Django Rest Framework (DRF) provides a variety of view classes that simplify the process of building RESTful APIs. These view classes are designed to handle different

HTTP methods (GET, POST, PUT, DELETE, etc.) and offer different levels of functionality and customization. Some of them are APIView, GenericAPIView, ListAPIView, ViewSet, ModelViewSet, etc.

- **APIView**

```
#for function-based views

@api_view([GET, POST, PUT, DELETE])
def fn_based_view(request):
      #code

#for class-based views

class classBasedView(APIView):
      #code
```

APIView allows developers to create fully customized API views by defining their own HTTP methods (e.g., get, post, put, delete) and implementing the corresponding logic. This flexibility is valuable when handling complex or unique API requirements.

# Quick Example for CRUD Implementation

### Creating a Model

```
# models.py
from django.db import models

class Task(models.Model):
    title = models.CharField(max_length=200)
    description = models.TextField()
    completed = models.BooleanField(default=False)

    def __str__(self):
        return self.title
```

### Creating the Corresponding Serializer

```
# serializers.py
from rest_framework import serializers
from .models import Task

class TaskSerializer(serializers.ModelSerializer):
    class Meta:
        model = Task
        fields = ('id', 'title', 'description', 'completed')
```

### CRUD implementation using Function based views

- **Create Task (POST request)**

```
@api_view(['POST'])
def create_task(request):
    serializer = TaskSerializer(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)
```

- **Retrieve List of Tasks (GET request)**

```
@api_view(['GET'])
def list_tasks(request):
    tasks = Task.objects.all()
    serializer = TaskSerializer(tasks, many=True)
    return Response(serializer.data, status=status.HTTP_200_OK)
```

- **Retrieve Task (GET request)**

```
@api_view(['GET'])
def retrieve_task(request, pk):
    try:
        task = Task.objects.get(pk=pk)
    except Task.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    serializer = TaskSerializer(task)
    return Response(serializer.data, status=status.HTTP_200_OK)
```

- **Update Task (PUT Request):**

```
@api_view(['PUT'])
def update_task(request, pk):
    try:
        task = Task.objects.get(pk=pk)
    except Task.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    serializer = TaskSerializer(task, data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.HTTP_200_OK)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

- **Delete Task (DELETE Request):**

```
@api_view(['DELETE'])
def delete_task(request, pk):
    try:
        task = Task.objects.get(pk=pk)
    except Task.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    task.delete()
    return Response(status=status.HTTP_204_NO_CONTENT)
```