**Serialization** in Django is the process of converting complex data types, like Django model instances, into native data types that can be easily rendered into JSON, XML, or other content types suitable for APIs. It is essential when building APIs because it allows data to be transformed into formats that clients (like mobile or web applications) can consume.

In Django, **Django REST Framework (DRF)** provides serializers, which help bridge the gap between Django models and JSON or other representations. This allows data to flow easily from the server to the client and vice versa.

**Why Use Serialization?**

1. **Data Exchange**: Serialization converts data into formats that can be transferred over a network.
2. **Data Validation**: Serializers can validate incoming data, ensuring that only valid data is saved to the database.
3. **Security**: By only exposing specific fields, serialization can help control what data is shared externally.
4. **Complex Data Handling**: Serializers simplify working with related data, nested data structures, and custom fields.

**How Serialization Works in Django REST Framework**

Django REST Framework offers several types of serializers to handle different use cases:

1. **ModelSerializer**: A shortcut for creating serializers that work directly with Django models.
2. **Serializer**: A more flexible option that requires explicitly defining each field and validation manually.

Project Name: class22simpleserialazer

AppName : simpleserapp

INSTALLED\_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    'simpleserapp',

    'rest\_framework',

]

#### Step 1: Define Your Model

**Start by creating a Django model that will represent the data structure you want to serialize.**

Models.py:

from django.db import models

# Create your models here.

class Book(models.Model):

    title = models.CharField(max\_length=100)

    author = models.CharField(max\_length=100)

    published\_date = models.DateField()

    isbn = models.CharField(max\_length=13)

    def \_\_str\_\_(self):

        return self.title

#### Step 2: Create a Serializer Class:s

1. **In a new file named serializers.py (in the same app as your model), create a serializer class for your model.**

**Import serializers from rest\_framework and your model.**

Serializers.py:

from rest\_framework import serializers

from .models import Book

class BookSerializer(serializers.Serializer):

    id =  serializers.IntegerField(read\_only = True)

    title =  serializers.CharField(max\_length = 100)

    author =  serializers.CharField(max\_length = 20)

    published\_date = serializers.DateField()

    isbn = serializers.CharField(max\_length=13)

    def create(self, validated\_data):

        return Book.objects.create(\*\*validated\_data)

    def update(self,instance, validated\_data):

          instance.title = validated\_data.get('title', instance.title)

          instance.author = validated\_data.get('author', instance.author)

          instance.published\_date = validated\_data.get('published\_date', instance.published\_date)

          instance.isbn = validated\_data.get('isbn', instance.isbn)

          instance.save()

          return instance

In Django REST framework (DRF), the create and update methods in a serializer are essential when you want to take in data, validate it, and then save it as a new or updated database record. Here’s a detailed breakdown of why these methods are necessary:

### . ****Purpose of**** create ****and**** update ****Methods in a Serializer****

* **create**: This method is responsible for creating a new instance of the model when you’re handling a **POST** request, which typically adds a new record.
* **update**: This method updates an existing instance, often used in **PUT** or **PATCH** requests, where you’re modifying an existing record.

### 2. ****How DRF Handles Data Submission with These Methods****

* When DRF receives data through an API request, it uses the serializer to **validate** the data before performing any database operation.
* The serializer doesn’t automatically know how to save this validated data to the database. That’s where the create and update methods come in:
  + **create**: Takes validated data and explicitly tells Django how to save it as a new model instance.
  + **update**: Uses validated data to update specific fields of an existing model instance.

### 3. ****Why You Can’t Skip These Methods****

* Without the create method:
  + The serializer would validate data but wouldn’t know how to save it as a new object.
  + You would get an error when you tried to call .save() on the serializer after validation.
* Without the update method:
  + The serializer wouldn’t know how to modify an existing object with the validated data.
  + This is essential for cases where you want to update only some fields, like with PATCH requests.

#### Step 3: Use the Serializer in Views

**In views.py, import the BookSerializer and use it in your views.**

Views.py:

from rest\_framework.response import Response

from rest\_framework.decorators import api\_view

from rest\_framework import status

from .models import Book

from .serializers import BookSerializer

# List all books (GET) or create a new book (POST)

@api\_view(['GET', 'POST'])

def book\_list(request):

    if request.method == 'GET':

        books = Book.objects.all()

        serializer = BookSerializer(books, many=True)

        return Response(serializer.data)

 **Book.objects.all()**: This retrieves all the records (books) from the Book model. The result is a queryset containing all Book instances from the database.

 **BookSerializer(books, many=True)**: The BookSerializer is used to convert the books queryset into a JSON representation. The many=True argument indicates that we are serializing multiple objects (a list of books).

**Response(serializer.data)**: This wraps the serialized data (a list of books) in a DRF Response object, which will be returned as the API response. By default, the Response object will return the data in JSON format.

    elif request.method == 'POST':

        serializer = BookSerializer(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)

**BookSerializer(data=request.data)**: The incoming data from the POST request (request.data) is passed to the BookSerializer. This data typically comes in the form of a JSON payload.

# Retrieve a single book by ID (GET)

@api\_view(['GET'])

def book\_detail(request, pk):

    try:

        book = Book.objects.get(pk=pk)

    except Book.DoesNotExist:

        return Response({"error": "Book not found."}, status=status.HTTP\_404\_NOT\_FOUND)

    serializer = BookSerializer(book)

    return Response(serializer.data)

# Update a book by ID (PUT)

@api\_view(['PUT'])

def update\_book(request, pk):

    try:

        book = Book.objects.get(pk=pk)

    except Book.DoesNotExist:

        return Response({"error": "Book not found."}, status=status.HTTP\_404\_NOT\_FOUND)

**BookSerializer(book, data=request.data)** initializes the serializer with two things:

* The book instance fetched from the database (this is the object that will be updated).
* The new data from the request.data (this is the data that the client sent to update the book). This data will typically be in JSON format, containing fields like title, author, etc.

    serializer = BookSerializer(book, data=request.data)

    if serializer.is\_valid():

        serializer.save()

        return Response(serializer.data)

    return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

# Delete a book by ID (DELETE)

@api\_view(['DELETE'])

def delete\_book(request, pk):

    try:

        book = Book.objects.get(pk=pk)

    except Book.DoesNotExist:

        return Response({"error": "Book not found."}, status=status.HTTP\_404\_NOT\_FOUND)

    book.delete()

    return Response({"message": "Book deleted successfully."}, status=status.HTTP\_204\_NO\_CONTENT)

app->urls.py

from django.urls import path

from .import views

urlpatterns = [

    path('books/', views.book\_list, name='book-list'),          # List and Create

    path('books/<int:pk>/', views.book\_detail, name='book-detail'),  # Retrieve single

    path('books/<int:pk>/update/', views.update\_book, name='book-update'),  # Update

    path('books/<int:pk>/delete/', views.delete\_book, name='book-delete'),  # Delete

]

Project\_> urls.py:

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

    path('api/', include('class22simpleserilazer.urls')),

]