Serializers in Django:

When you build APIs in Django using **Django REST Framework**, serializers help you convert data between:

Python/Django Objects	⇄	JSON / API Response / Request Body
QuerySets, Model Instances	⇄	JSON for API calls

Types of Serializers in DRF

1) `Serializer` (Manual Approach)

You define fields manually.

```
from rest_framework import serializers

class ProductSerializer(serializers.Serializer):
    id = serializers.IntegerField()
    name = serializers.CharField()
    price = serializers.FloatField()
```

---

2) `ModelSerializer` (Auto builds from Django models)

Takes a model and builds serializer fields for you.

```python
from rest_framework import serializers
from .models import Product

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = '__all__'  # or list: ['id', 'name', 'price']
```

This is the one used 90% of the time because it's faster and clean.

Difference: Serializer vs ModelSerializer (to explain in class)

Feature	Serializer	ModelSerializer
Manual fields	Yes	No
Based on model	No	Yes
Custom data not in models	Easy	Harder
CRUD helpers (`save()`)	Manual	Built-in

Basic Example — CRUD with Product ModelSerializer

```python
# models.py
class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.FloatField()
```

```python
# serializers.py
class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = ['id','name','price']
```

CREATE

```
import json
from django.http import JsonResponse
from .serializers import ProductSerializer


def create_product(request):
    if request.method != 'POST':
        return JsonResponse({'detail':'Method not allowed'}, status=405)

    data = json.loads(request.body)
    sr = ProductSerializer(data=data)
    if sr.is_valid():
        prod = sr.save()
        return JsonResponse(ProductSerializer(prod).data, status=201)
    return JsonResponse(sr.errors, status=400)
```

READ ONE

```
from django.http import JsonResponse
from .models import Product
from .serializers import ProductSerializer


def get_product(request,id):
    p = Product.objects.get(id=id)
    return JsonResponse(ProductSerializer(p).data, safe=False)
```

READ MANY

```
def list_products(request):
    qs = Product.objects.all()
    return JsonResponse(ProductSerializer(qs, many=True).data, safe=False)
```

UPDATE

```
def update_product(request,id):
    if request.method not in ('PUT','PATCH'):
        return JsonResponse({'detail':'Method not allowed'}, status=405)

    p = Product.objects.get(id=id)
    data = json.loads(request.body)
    partial = request.method=='PATCH'
    sr = ProductSerializer(p, data=data, partial=partial)
    if sr.is_valid():
        p = sr.save()
        return JsonResponse(ProductSerializer(p).data)
    return JsonResponse(sr.errors, status=400)
```

Fields control (`fields`, `read_only_fields`, `extra_kwargs`)

What it does

- `fields` lists which model fields the serializer exposes.
- `read_only_fields` marks fields that should be present in output but not accepted on input.
- `extra_kwargs` lets you set per-field options (e.g. `required`, `default`, `write_only`, `validators`, `help_text`).

```python
class StudentSerializer(serializers.ModelSerializer):
    class Meta:
        model = Student
        fields = ['id', 'name', 'age']
        read_only_fields = ['id']
        extra_kwargs = {
            'age': {'required': False, 'default': 18}
        }
```

Behavioral notes

- If `age` is omitted in input, serializer will use the `default` during validation if the field is not required.
- `read_only_fields` makes the field ignored for `create()` / `update()` — incoming data containing `id` will be ignored (it won't raise error).
- `write_only` (via `extra_kwargs`) is useful for things like `password` (show on input, not in serialized output).

Pitfalls

- `default` only applies if the field is missing; if the field is present but null and `allow_null=False`, validation will fail.
- `read_only_fields` only prevents writing through the serializer API — database-level defaults/constraints still apply.

When to use

- Use `fields` to limit exposure (security/contract).
- Use `read_only_fields` for auto-generated fields (`id`, `created_at`).
- Use `extra_kwargs` to tweak per-field validation without writing custom fields.

✅ Write-Only Field Example (Password Case)

python Copy code

```python
# serializers.py
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User   # assume django.contrib.auth.User
        fields = ['id', 'username', 'password']
        extra_kwargs = {
            'password': {'write_only': True}
        }
```

Meaning:

Field	Can be Given in Input (POST/PUT/PATCH)	Will Appear in Output (serializer.data)
`username`	✅ yes	✅ yes
`password`	✅ yes (write allowed)	❌ no (hidden in output)
`id`	❌ no (auto)	✅ yes

6) Custom validation

There are two common kinds of serializer validation:

Field-level validator — `validate_<fieldname>(self, value)`

Runs for a single field after built-in field validation.

python Copy code

```python
def validate_age(self, value):
    if value < 5:
        raise serializers.ValidationError("Too young")
    return value
```

Object-level validator — `validate(self, attrs)`

Runs after individual fields validated; good for cross-field logic.

python Copy code

```python
def validate(self, attrs):
    if attrs['age'] < 10 and attrs['name'] == 'Admin':
        raise serializers.ValidationError("Invalid student profile")
    return attrs
```

Examples & notes

- Field-level validators receive the *already-parsed* value for that field.
- Object-level can inspect multiple fields together and should return the (possibly modified) `attrs`.
- Raise `serializers.ValidationError` with either message string, list, or dict for field-specific errors.

Pitfalls

- If you mutate `attrs` inside `validate`, remember to return it.
- For raising a field-specific error inside `validate`, structure the error as `{'field_name': 'error'}` (or the serializer will show it as non-field error).

When to use

- Use field-level for single-field constraints (min length, range).
- Use object-level for business rules involving multiple fields.